Author:
Soria-Vázquez, Eduardo

Title:
Towards Secure Multi-Party Computation on the Internet

Few Rounds and Many Parties

Towards Secure Multi-Party Computation on the Internet: Few Rounds and Many Parties

By

EDUARDO SORIA VÁZQUEZ

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

OCTOBER 2018

Word count: 74,000 words

A mi padre.

Abstract

Multi-Party Computation (MPC) protocols allow a set of mutually distrustful parties to securely compute on their joint private inputs, maintaining the privacy of those. Since its early conception in the eighties, a plethora of research as well as some real world deployments have taken place, demonstrating the relevance of such an interesting mathematical problem. Nevertheless, most of the research and all deployments have focused on scenarios where both the number of parties and network latency are very low. This thesis lays out the first steps towards practical deployments of large-scale MPC, where many parties may be involved and they might only be connected through a high-latency network such as the Internet.

Both theoretical and empirical evidence shows that non-constant-round protocols cannot perform well on slow networks for many functions. In order to overcome this issue, all the works in this thesis include concretely efficient constant-round protocols based on multi-party garbled circuits. Chapter 3 and Chapter 4 deal with the very strong setting where an active adversary corrupts all but one parties. The former achieves such goal by using homomorphic encryption for low-depth circuits. The latter, based on subsequent work, improves the theoretical understanding of the problem and utilises oblivious transfer to improve efficiency.

In Chapter 5 we provide both constant and non-constant round protocols that can handle large numbers of parties more efficiently. We leverage our performance improvements by relaxing the setting where all but one parties are corrupted to one where a small minority of honest participants can be assumed. Concretely, such change allows to distribute secret key material so that each party only holds a 'short' part of the key. Security is then based on the concatenation of all honest parties' keys rather than on each party's individual key, improving both the communication and computation complexities.

Acknowledgements

The last three years have marked an important period of my life, both personally and as a researcher. They would have not been the same without the support of family, friends and colleagues. I would like to start by thanking Nigel Smart for giving me this opportunity, being my advisor and introducing me to others in the community. Martijn Stam also deserves a mention in these grounds for his help and advice, specially during the last year.

Science is a collective enterprise. I would not be writing these lines in this precise moment if I did not get to work with my co-authors Carmit Hazay, Marcel Keller, Yehuda Lindell, Emmanuela Orsini, Dragoş Rotaru, Peter Scholl, Nigel Smart and Srinivas Vivek. I have learned a great deal from these collaborations, as well as from general chats about cryptography and research. These chats have, of course, not been limited neither to those topics, nor to my co-authors. I can only be grateful for all the nice people and the human aspects of the working environment at University of Bristol – lunches, coffees and Friday pubs included.

Last, but not least, much of the work and enjoyable conversations would not have taken place without the support of the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 643161, in more familiar terms, ECRYPT-NET. I can speak for at least many of us the Early Stage Researchers when I say this project has provided an excellent framework to develop our research and to learn about the many aspects of cryptography, beyond the purely academic. It has been a true pleasure to also share this journey with the rest of the programme members!

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

'*Knowledge is power*'. Even though there are previous documented occurrences of the sentence [5, 115], the disputed aphorism is usually miss-attributed to Sir Francis Bacon, one of the main figures in the empiricist tradition. Following Bacon's views [13], knowledge comes through observation, formulation of hypothesis and amassing data. Once such data is analysed, hypothesis are then confirmed, refined or rejected. More evolved forms of Bacon's method have been applied to verify the truthfulness of the 'knowledge is power' saying – as for example in the *asymmetry of information* concept of economics [2] – but if we conversely examine the methodology through the proverb's lenses, we can conclude that data gathering and processing are sources of power.

The contents of this thesis are related to the field of Cryptography and not (directly, at least) to Epistemology or Political Science. Nevertheless, as it was simply put in Philip Rogaway's distinguished lecture in Asiacrypt 2015 [119], cryptography rearranges power: It determines *who* can gather *which* data, and *how* such data can be processed. Rogaway was not the first one to notice this fact. Whitfield Diffie, broadly considered a pioneer in the modern science of cryptography, recalled during a trial [102, 119] a conversation with his wife in 1973:

> *'I told her that we were headed into a world where people would have important, intimate, long-term relationships with people they had never met face to face. I was worried about privacy in that world, and that's why I was working on cryptography'.*

Thanks to Diffie's foundational work together with Hellman [52] (as well as that of many other authors in the subsequent decades), citizens, businesses and organizations can now enjoy secure storage and communication systems. When those are properly deployed, they address the issue of *who* can collect *which* data, i.e. the desired sender and recipient(s) of the information. But is this always enough? What about *how* is that data used?

It is now part of the folklore that businesses, states and others collect and hold vast amounts of information about people and their surroundings. Such data creates infrastructures which mediate or even lead, with varying degrees of fairness, numerous aspects of our daily lives: Whether we have access to a loan or insurance, what we get to see in social media or search engines, what healthcare treatments we need, etc. The list could go on for pages. On the other hand, there is a rising concern about the *unintended* use – from the perspective of the affected individual, or society more broadly – of collected data by the *intended* recipients. This worry seems to be strengthened by the former routine of amassing data, which a lot of times happens with little transparency about why different pieces are gathered, or in the worst cases without any consent at all.

Proof that the debate around privacy and data mistreatment has become a very important one is the introduction of laws such as the European Union's General Data Protection Regulation (GDPR) [55]. The new legislation, which started being enforced as recently as May 2018, takes a tiered approach to the different levels of negligence or misuse. Notoriously and for the most serious infringements, organizations can be fined up to twenty million euros or 4% of their total worldwide turnover, whichever is higher [55, Art. 83(5)].

Notwithstanding that policy is part of the solution here, on the purely technical side there seems to remain an insurmountable dilemma. Can we really control *how* our information is used? Computing on data from multiple sources intuitively requires to *see* their data. Thus, once sources contribute their information, they have to give up on all privacy and control over it. Surprisingly as it may seem to outsiders of the field, cryptography has also a solution to this issue, which is the topic of this thesis: *Multi-Party Computation* (MPC).

1.1 A Brief History of Practical Multi-Party Computation

Multi-Party Computation allows a set of parties to jointly compute a mutually agreed function on their data, while keeping their inputs private. Security of MPC guarantees that only the *intended* result becomes known, even when a coalition of the participants try to attack the system. Thus, using cryptography we can, in fact, control *who* computes on information of *our choice* and *how*, being even able to keep that departing information secret!

Diffie and Hellman foresaw that the development of cheap hardware and the reduction in cost of cryptographic devices would lead to applications requiring new (in their case, public-key) cryptographic systems [52]. No such prediction is known to have been stated by them about Multi-Party Computation, but for the same reasons, as well as a due to a vast body of research in the area, MPC is also recently becoming an important system deployed in the real world.

The first such deployment was secure computation of auctions, in order to determine the market clearing price for sugar beet production contracts in Denmark [31]. The scenario included on the one hand Danisco, holding a monopoly on the sugar beet processing in the country and, on

the other hand, 1,229 farmers supplying the sugar beets to Danisco. Whereas auctions are easy to solve when a trusted third party can be found to handle the bids and, in some cases as this one, keeping them secret, no such entity could be agreed upon by all participants due to conflicting interests. Back in January 2008, the nation-wide computation was carried out by three parties and took around half an hour.

Other applications that have been either proposed or successfully carried out in real scenarios include tax fraud detection [29], sociological and medical studies [30, 77] and network analysis [7]. An additional general application of MPC is that of distributing trust by decentralization. Storing or processing information in the clear in a single place constitutes a single point of failure for adversaries trying to break a system. Data breaches and ransomware are frequent in the news, sometimes with terrible consequences. By substituting such single points of failure with a set of multiple servers or entities running MPC, the resulting system becomes more robust: Attackers require to take and hold control over different parties running MPC before being detected and repelled.

There is something all the proposals above have in common: Few parties are assumed to be involved in the computation, and a very low latency network connects them. Notably and for the first deployment of MPC [31], even though the figure of a trusted auctioneer was removed, each of the 1,229 bidders had to trust at least two of the three parties carrying out the computation to be honest. Moreover, they still had to be confident that the parties they considered untrustworthy would follow the protocol instructions. Finally, all of these assumptions had to be combined with the fact that all computing parties were very closely located: They were, in fact, connected through an Ethernet LAN [31].

A huge progress in protocol design has taken place since the first secure auction was run in Denmark a decade ago. MPC can now run significantly faster even against *active* adversaries (i.e., those who arbitrarily deviate from the protocol description) that can be assumed to control a majority of the computing parties. Still, most of *multi-party* protocols do not scale so well with respect to neither network latency nor the number of participants. For the former issue, the primary cause is the *round complexity* of protocols, specially when the function to compute is represented by deep circuits [25]. This thesis lays out the first steps in the long road towards practical deployments of large-scale MPC, where many parties may be involved and they might only be connected through a high-latency network such as the internet.

1.2 Outline of this Thesis

The rest of this dissertation will be structured as follows. In Chapter 2 we review some basic notions related to Multi-Party Computation and cryptography more in general. Within those pages, we also focus our attention on the more advanced notions of Universally Composable security (UC) [35] and garbled circuits. The former concept, which has become the standard

framework for proving security in MPC, will be a very useful tool during the rest of the thesis. The UC model not only provides strong security guarantees, but it also helps to simplify the description of multi-party protocols by dividing them into more intuitive, smaller building blocks. Such divisions will be needed when presenting the latter garbled circuits technique, which we first describe in the simpler, two-party case. The subsequently introduced multi-party scenario, following the work [18] of Beaver, Micali and Rogaway (BMR), is the most important concept for the comprehension of Chapters 3 and 4, as well as an important part of Chapter 5.

Chapter 3 describes how to perform the *garbling* phase of BMR – one of the simpler building blocks we can describe on the UC framework – using the notion of homomorphic encryption, which became a significant improvement on the state of the art efficiency for this step over previous works. The following Chapter 4 gives a general transformation from any MPC protocol for boolean circuits to BMR, as well as a concrete, more efficient instantiation using the TinyOT protocol. The contents presented there have become especially relevant not only in terms of efficiency, but also conceptually, as several other works now use it in order to build and prove their protocols secure.

While the two precedent chapters consider a setting where an adversary controls all but one of the parties, Chapter 5 explores how efficiency can be improved when an *arbitrary* number of parties are free from such control. Whereas previous works in MPC consider mostly very particular thresholds of parties corrupted by the adversary (e.g. less or more than a half, or a third), we parametrize protocols within the dishonest majority setting in the number of honest parties. Such parametrization is specially relevant in scenarios where many parties (tens, or even hundreds) are running MPC, as it is not too realistic to consider, for example, that nine hundred ninety nine parties are colluding against a single honest party. We leverage this parametrization to increase the efficiency of both secret-sharing based and garbled circuit-based protocols, in which we shorten the length of honest parties' symmetric keys below the usual computational security parameter. Whereas looking at those keys *individually* would result in insecure protocols, security in our case relies on the *concatenation* of all parties' keys, hence leading to secure computation on the whole.

We conclude in Chapter 6, where we synthesize the impact of our results in the areas described above, as well as we point to interesting open questions and directions towards MPC for the masses.

1.3 Contributions of the Author

The contents of this thesis are based on the following publications during my doctoral studies:

[96] Y. LINDELL, N. P. SMART, AND E. SORIA-VAZQUEZ, *More efficient constant-round multi-party computation from BMR and SHE*, in TCC 2016-B, Part I, M. Hirt and A. D. Smith, eds., vol. 9985 of LNCS, Springer, Heidelberg, Oct. / Nov. 2016, pp. 554–581.

[70] C. HAZAY, P. SCHOLL, AND E. SORIA-VAZQUEZ, *Low cost constant round MPC combining BMR and oblivious transfer*, in ASIACRYPT 2017, Part I, T. Takagi and T. Peyrin, eds., vol. 10624 of LNCS, Springer, Heidelberg, Dec. 2017, pp. 598–628.

[69] ——, *TinyKeys: A new approach to efficient multi-party computation*, in CRYPTO 2018, Part III, H. Shacham and A. Boldyreva, eds., vol. 10993 of LNCS, Springer, Aug. 2018, pp. 3–33.

Unless otherwise specified in the relevant parts of the document, the results presented are due to equal collaboration by all co-authors. During the same period, I have also published the following works, not included here:

[79] M. KELLER, E. ORSINI, D. ROTARU, P. SCHOLL, E. SORIA-VAZQUEZ, AND S. VIVEK, *Faster secure multi-party computation of AES and DES using lookup tables*, in ACNS 17, D. Gollmann, A. Miyaji, and H. Kikuchi, eds., vol. 10355 of LNCS, Springer, Heidelberg, July 2017, pp. 229–249.

[68] C. HAZAY, E. ORSINI, P. SCHOLL, AND E. SORIA-VAZQUEZ, *Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT)*, in ASIACRYPT 2018, Part III, T. Peyrin and S. D. Galbraith, eds., vol. 11274 of LNCS, Springer, Dec. 2018, pp. 86–117.

An obvious but understated truth is that ideas that do *not* succeed are also part of a PhD. As everyone else, I have had my share of those, as well as having other unpublished works in the making that hopefully will not fall into that category.

CHAPTER 2

PRELIMINARIES

The goal of this chapter is to cover the technical background that will be needed for the understanding of the following chapters. We start by refreshing some basic notions and cryptographic primitives, after which we discuss the diverse goals of Multi-Party Computation (MPC) protocols and the different ways of measuring their efficiency. Following this presentation, we delve into the details of proving security of cryptographic primitives and protocols according to the Universal Composability (UC) framework, which has become the standard model of security in the area of MPC.

We conclude the chapter by giving more details about garbled circuits and their role in both Two-Party and Multi-Party Computation. We start by focusing on the two-party case, where garbled circuits constitute the most efficient solution at the moment. This serves as an introduction to the more complex MPC scenario, for which we describe the protocol of [18] by Beaver, Micali and Rogaway (BMR). Their seminal construction spurred a line of work which is also the starting point for the contents of Chapters 3 and 4, as well as Section 5.4 in Chapter 5.

2.1 Prerequisites

Throughout this thesis, we assume familiarity with some basic linear algebra and probability concepts. In this section we give a quick recap of some of them, as well as other essential cryptographic notions. The first notion we want to refresh is that of negligibility:

Definition 2.1. A function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* if for every positive polynomial p there exists $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$:

$$f(n) \leq \frac{1}{p(n)}$$

This puts us in a good position to define the different notions of indistinguishability between *distribution ensembles*, i.e. countable sets of probability distributions. These rely on the concept of a *distinguisher* $\mathscr{D}$ which, given a string, outputs '0' if it believes it has been sampled according to distribution X or, otherwise, it outputs '1' believing it comes from distribution Y.

Informally, *computationally* indistinguishable constructions are secure against distinguishers whose computation and storage resources are bounded – to, say, not exceeding the total amount of those that are estimated to be available on Earth at the moment. *Statistical* security removes that constraint from the distinguisher and reduces security to a negligible advantage in lucky guessing, whereas *perfect* or *unconditional* security is the strongest notion, which means that the random variables are identically distributed. More formally, we have the following definitions:

Definition 2.2. Let $X = \{X(a,\kappa)\}_{a\in\{0,1\}^*,\kappa\in\mathbb{N}}$ and $Y = \{Y(a,\kappa)\}_{a\in\{0,1\}^*,\kappa\in\mathbb{N}}$ be two distribution ensembles. We say that X and Y are *computationally indistinguishable*, denoted $X \overset{c}{\approx} Y$, if for every probabilistic polynomial-time distinguisher $\mathscr{D}$ and every $a \in \{0,1\}^*$, there exists a negligible function negl such that:

$$\left|\Pr_{x\leftarrow X(a,\kappa)}\left[\mathscr{D}(x,a,1^\kappa)=1\right] - \Pr_{y\leftarrow Y(a,\kappa)}\left[\mathscr{D}(y,a,1^\kappa)=1\right]\right| < \mathsf{negl}(\kappa),$$

where κ is denoted the *computational security parameter* and $\mathscr{D}$ is given the unary input 1^κ so it can always run in time polynomial in κ.

Definition 2.3. Let $X = \{X(a,s)\}_{a\in\{0,1\}^*,s\in\mathbb{N}}$ and $Y = \{Y(a,s)\}_{a\in\{0,1\}^*,s\in\mathbb{N}}$ be two distribution ensembles. We say that X and Y are *statistically indistinguishable*, denoted $X \overset{s}{\approx} Y$, if for every distinguisher $\mathscr{D}$ and every $a \in \{0,1\}^*$, there exists a negligible function negl such that:

$$\left|\Pr_{x\leftarrow X(a,s)}[\mathscr{D}(x,a)=1] - \Pr_{y\leftarrow Y(a,s)}[\mathscr{D}(y,a)=1]\right| < \mathsf{negl}(s),$$

where s is denoted the *statistical security parameter*.

Definition 2.4. Let $X = \{X(a)\}_{a\in\{0,1\}^*}$ and $Y = \{Y(a)\}_{a\in\{0,1\}^*}$ be two distribution ensembles. We say that X and Y are *perfectly* or *unconditionally indistinguishable*, denoted $X \equiv Y$, if for every distinguisher $\mathscr{D}$ and every $a \in \{0,1\}^*$:

$$\left|\Pr_{x\leftarrow X(a)}[\mathscr{D}(x,a)=1] - \Pr_{y\leftarrow Y(a)}[\mathscr{D}(y,a)=1]\right| = 0.$$

2.2 Primitives

We turn to describe several cryptographic primitives that will be useful across the different chapters of this thesis.

2.2.1 Secret Sharing

Secret sharing was introduced by Shamir in 1979 [122] as a way of securely dividing secret data X within a set of mutually distrusting parties. The trusted dealer distributes pieces (called *shares*) of the data in such a way that only authorised subsets of parties can reconstruct the original secret, while the unauthorised subsets cannot learn anything about X. Imagine a situation where someone wants to leave a will and testament without resorting to a public notary. That person could then, for example, secret share the document between her loved ones in such a way that the only way they could learn anything about what was written is by putting together all of their shares.

In this thesis we are only concerned about secret sharing schemes for which the determining factor about whether a subset of parties is authorised or not is its size. More concretely, a subset of parties is authorised if it is consist of at least t of them, which in the previous example would be $t = n$ where n is the number of legatees.

Shamir's construction [122] is probably the most famous scheme. First, the dealer chooses a random degree $t-1$ polynomial in such a way that the constant term is equal to the secret X. Afterwards, each party is privately distributed an evaluation of the polynomial on a publicly known point in a finite field. Hence, X can be recovered if and only if any subset of at least t parties pools together their evaluations of the original polynomial, so they can reconstruct it via interpolation and learn the constant term.

Additive secret sharing has a fixed threshold of $t = n$ and is the most efficient scheme for that value. To share a secret $X \in \mathbb{Z}_m$, the dealer randomly samples $x_i \in \mathbb{Z}_m, i \in [n-1]$ uniformly at random, and sets $x_n = X - \sum_{i=1}^{n-1} x_i$. Security follows as up to any $n-1$ shares of X its value is hidden by the missing, uniformly random share which acts as a one-time-pad.

2.2.2 Pseudorandom Functions

Pseudorandom functions (PRFs) are one of the most important objects in modern cryptography. They efficiently and deterministically stretch finite-length keys into arbitrary length strings that are computationally indistinguishable from uniform. Whereas generating actual random bits is a difficult and costly process, the use of PRFs allows cryptographers to assume that parties have access to an unlimited pool of random data, as that problem is reduced to that of sampling the PRF key. We will make extensive use of PRFs in the following chapters, for which we provide the precise definition below:

Definition 2.5. Let $F : \{0,1\}^n \times \{0,1\}^n \mapsto \{0,1\}^n$ be an efficient, length preserving, keyed function. F is a *pseudorandom function* if for all probabilistic polynomial-time distinguishers $\mathscr{D}$, there exists a negligible function negl such that:

$$\left| \Pr[\mathscr{D}^{F_k(\cdot)}(1^\kappa) = 1] - \Pr[\mathscr{D}^{f(\cdot)}(1^\kappa) = 1] \right| \leq \mathsf{negl}(\kappa),$$

where the first probability is taken over the randomness of $\mathscr{D}$ and the key $k \in \{0,1\}^n$ and the second one is over the choice of f from all functions mapping $\{0,1\}^n \mapsto \{0,1\}^n$, as well as the randomness of $\mathscr{D}$.

2.2.3 Oblivious Transfer

Oblivious Transfer (OT) protocols were introduced by Rabin [113] and Wiesner [130] and run between two mutually distrusting parties – sender and receiver. OT guarantees that the sender can transmit part of their input while remaining oblivious to which one and in such a way that the receiver learns no information on the rest of said input. In its most basic form, denoted 1-out-of-2 OT, the sender has inputs m_0, m_1, while the receiver has an input bit b. At the end of the protocol the receiver learns m_b, but nothing about m_{1-b}, and the sender learns nothing about b.

From a theoretical perspective, OT is known to be both necessary and sufficient to realize general MPC [82]. On the practical side, although Oblivious Transfer requires the use of public-key cryptography [73], it is possible to 'extend' a initial batch of OTs computed in such way to produce many more OTs using just symmetric-key cryptography [10, 17, 51, 74]. This can be interpreted as the OT equivalent of hybrid encryption, where parties encrypt a large amount of data using symmetric cryptography and the symmetric key is encapsulated in a public-key cryptosystem. These extension techniques lead to highly efficient MPC protocols as the ones we will describe in Chapter 4 and Chapter 5.

2.3 Multi-Party Computation

Picture a set of parties $P_1, \ldots, P_n$ each of which has a secret input $x_1, \ldots, x_n$. The parties want to compute some function of their joint inputs $y = f(x_1, \ldots, x_n)$, however, they do not trust each other and they want to keep their inputs private. If the parties could agree on some trusted third party $\mathscr{F}$, then they would just have to hand their data to $\mathscr{F}$, who would compute the function on their behalf and send them their prescribed part of the result y. A Multi-Party Computation (MPC) protocol is said to securely compute f if running it achieves exactly the same guarantees this ideal entity $\mathscr{F}$ would provide.

How should one then choose an MPC protocol amongst those providing the same set of guarantees? The practical answer would be just picking the most *efficient* one, but the choice is not as straightforward as it could first seem. The most important aspects to consider when making this decision are what can be assumed about both the behaviour of the parties involved in the computation, and the channels over which they communicate. For all the possible combinations of the concepts we are about to enumerate, the protocol to pick would be a different one. A non-exhaustive but more complete overview of these can be found in [109], as well as how these assumptions relate to each other and the feasibility of MPC.

All misbehaving parties are considered to be colluding and we treat them as a single entity called the *adversary*. *Passive* adversaries (also referred to as *semi-honest* adversaries) follow all the steps as prescribed by the MPC protocol, whereas *active* (or *malicious*) adversaries may arbitrarily deviate from it in an attempt to breach security. The rest of the parties in the protocol are referred to as *honest*. A *static* adversary chooses which parties to corrupt before the start of the protocol, whereas an *adaptive* adversary can do so over the course of the execution, taking advantage of the information obtained from already corrupted parties. Finally, the adversary is also bound not to corrupt more parties than a given *threshold*, for example a minority of the parties or all but one of them.

A majority of the MPC literature assumes that the communication network is *synchronous*, i.e. that all messages sent during the protocol execution arrive timely and in an expected order. When this cannot be assumed we say the protocol is secure in *asynchronous* networks that the adversary can tamper with. In this thesis we focus on *synchronous* networks, within which there are still relevant distinctions to be made in practice: Does the network have a high latency? What is its bandwidth?

Modern, practical MPC protocols typically fall into two main categories: those based on secret-sharing [28, 45, 47, 64, 76, 114], and those based on garbled circuits [18, 41, 86, 90–92, 101, 132]. Secret-sharing based protocols tend to have lower communication requirements in terms of bandwidth, but require a large number of rounds of communication, which increases with the complexity of the function. In this approach the parties first secret-share their inputs and then evaluate the circuit gate by gate while preserving privacy and correctness. In low-latency networks, they can have an extremely fast online evaluation phase, but the round complexity makes them much less suited to high-latency networks (e.g., when the participating parties are on opposite sides of the world), where protocols with many rounds perform very poorly [25, 121].

Most of the research effort on making secure computation practically efficient has focused on the case of two parties [104, 111, 131]. Protocols based on Yao's garbled circuits [131] have achieved extraordinary efficiency both against passive [10, 20] and active adversaries [71, 88, 90, 92, 94, 95, 123]. In contrast, secure computation protocols for an arbitrary number of parties are way behind. When considering the secret-sharing based approach, variants of GMW can be used against passive adversaries [39, 64], whereas active adversaries can be countered using the protocols of SPDZ and TinyOT [44, 47, 87]. However, as mentioned above, these protocols have inherent inefficiency based on the fact that the number of rounds in the protocol is linear in the *depth* of the circuit that the parties compute.

In contrast to the impressive progress made in garbled circuits for two-party computation, very little was achieved for their operation in the multi-party setting. We will review the basics of Yao's garbled circuits in Section 2.5, which will serve as a baseline to compare them with their multi-party counterpart in Section 2.6. Before doing so, we next turn our attention to the details of meaningfully defining security for the more advanced cryptographic constructions to come.

2.4 Universally Composable Security

Throughout the thesis we will prove the security of our protocols following the Universal Composability (UC) framework [35] or, better said, a simplified variant thereof [36]. We do not attempt to formally define the whole framework in all details, for which we refer the reader to the original paper [35], but to give an overview of the main concepts, properties and definitions.

The core idea is that of comparing any execution of a given protocol π to an execution where the task is performed instead by a third party $\mathscr{F}$, called the *functionality*, which is trusted by everyone. We call the former scenario the *real world* and the latter the *ideal world*, reflecting that it represents what we wish for the protocol to achieve. The comparison duty is assigned to the *environment*, $\mathscr{Z}$, which epitomises everything else happening besides the protocol execution. Finally, the original protocol is deemed secure if no environment – read, no situation under which the protocol is run – can distinguish between both worlds.

Dealing as we are with security, the framework would not be complete without a *real-world adversary* $\mathscr{A}$ attacking the protocol, or what is the same, trying to ease the environment's job of telling apart the real execution from the ideal functionality. Any such adversary has a counterpart in the ideal world, which we call the *simulator* $\mathscr{S}$.

More formally, all entities introduced above – parties, adversary, simulator, functionality and environment – are modelled as Interactive Turing Machines (ITM) running in probabilistic, polynomial time (PPT). Precise details on what it means for a Turing machine to be interactive can be found in [35, 36], but the basic intuition is that two new types of tapes are added to the machine: read-only *incoming communication tapes*, and write-only *outgoing communication tapes*.

The real world is conformed by n ITMs $P_1,\ldots,P_n$ running the protocol π on behalf of the parties. We denote by A the subset of them *corrupted* by the adversary, $\mathscr{A}$. When a party P_i is corrupted by a passive adversary, $\mathscr{A}$ gets to read the internal status of that party, including the content on all its tapes. If the protocol is instead run in the presence of an active adversary, then corruption means that $\mathscr{A}$ additionally takes over the execution of P_i.

The ideal world consists of the ideal functionality $\mathscr{F}$, dummy parties $\tilde{P}_1,\ldots,\tilde{P}_n$ and the simulator $\mathscr{S}$. $\mathscr{F}$ is an ITM performing the desired task of the real world protocol π. When a party has been corrupted, $\mathscr{F}$ may treat it differently from an honest one by sending it some additional information. Dummy parties just relay inputs from the environment $\mathscr{Z}$ to the functionality $\mathscr{F}$ and, conversely, outputs from $\mathscr{F}$ to $\mathscr{Z}$. The simulator is an ITM interacting with $\mathscr{F}$ which tries to produce a transcript of the communication between parties that matches up the real world execution with the adversary $\mathscr{A}$.

As it has been hinted before, the environment $\mathscr{Z}$, also an ITM, is the fundamental component of the UC framework. It has total control over the adversary, it can choose the inputs and read the outputs of *all* parties, and it further sees everything sent between the parties in the real world. Notably, all of this means that $\mathscr{Z}$ is an 'interactive distinguisher'. This arises from the fact that

$\mathcal{Z}$ can exchange information with the adversary at any point during the execution, in particular after any message is sent. For this reason, the simulator's task of generating a transcript of the communications is specially hard, as it has to do so while interacting with the environment.

Finally, in order to argue about indistinguishability, let $\mathbf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the distribution ensemble representing the output of the environment $\mathcal{Z}$ when interacting with an adversary $\mathcal{A}$ and parties $P_1,\ldots,P_n$ running protocol π. In a similar way, let $\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ represent the output of $\mathcal{Z}$ when interacting with the simulator $\mathcal{S}$ and the ideal functionality $\mathcal{F}$. Security of a protocol in the UC framework is then defined as follows.

Definition 2.6 (UC security)**.** A protocol π realizes an ideal functionality $\mathcal{F}$ with perfect, statistical or computational security if for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{Z}$, it holds that

$$\mathbf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}},$$

where $\approx$ represents the relevant notion between perfect, statistical and computational security (see Section 2.1).

The most important feature of UC security, as its name states, is *composability*. Intuitively, this means that if we have proved that protocol ρ securely realizes a functionality $\mathcal{G}$, both can be mutually interchanged for the purpose of building a protocol π realizing some more advanced functionality $\mathcal{F}$. When the protocol π is defined using functionality $\mathcal{G}$, we say that we are in the *$\mathcal{G}$-hybrid model*. This hybrid is similar to the real world, except in addition to standard messages amongst themselves, parties may also interact with any number of local copies of $\mathcal{G}$ throughout the course of π.

Theorem 2.4.1 (Universal Composition Theorem [35])**.** *Let ρ be a protocol securely realizing a functionality $\mathcal{G}$. Next, let π be a protocol securely realizing another functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Then for every real world adversary $\mathcal{A}$ there exists a hybrid model adversary $\tilde{\mathcal{A}}$ such that for every environment $\mathcal{Z}$*

$$\mathbf{REAL}_{\pi_\rho,\mathcal{A},\mathcal{Z}} \approx \mathbf{HYB}^{\mathcal{G}}_{\pi,\tilde{\mathcal{A}},\mathcal{Z}},$$

where π_ρ is the protocol π modified so that calls to $\mathcal{G}$ are replaced with a subroutine call to protocol ρ.

As it was mentioned earlier, we do not need the full power and flexibility of the UC framework for the work exposed in this thesis. Common assumptions across all our protocols, which simplifies their proofs, are the following. First, we only deal with *static* adversaries, which means that the set A of corrupted parties is fixed at the start of execution. Second, all communication is done over authenticated channels, meaning that the adversary cannot change the messages sent by honest parties. Finally, we assume synchronous communication between all parties, which means

The General MPC Functionality: $\mathcal{F}_{\text{MPC}}$

The functionality is parametrized by a function $f(x_1,\ldots,x_n)$, which is input as a binary circuit C_f. It consists of three externally exposed commands **Initialize**, **InputData**, and **Output** and one internal subroutine **Wait**.

Initialize: On input ($init$, C_f) from all parties, where C_f is a Boolean circuit, the functionality activates and stores C_f.

Wait: The functionality waits on the adversary for a reply. If $\mathcal{A}$ returns $\perp$ then the functionality aborts, otherwise it continues.

InputData: On input ($input$, P_i, varid, x_i) from P_i and ($input$, P_i, varid, ?) from all other parties, with varid a fresh identifier, the functionality stores (varid, x_i). The functionality then calls **Wait**.

Output: On input ($output$) from all parties, if (varid,x_i) is stored for each P_i, the functionality computes $y = f(x_1,\ldots,x_n)$ and outputs y to the adversary. The functionality then calls **Wait**. If **Wait** does not result in an abort, the functionality outputs y to all parties.

Figure 2.1: The MPC Functionality: $\mathcal{F}_{\text{MPC}}$

that the protocol proceeds in *rounds*. During each round, every party may send one message to each of the other parties, which will be delivered by the end of the round.

There have been different proposals about how to model synchronous networks in the UC model. Canetti [35] advocates for the use of a $\mathcal{F}_{\text{Syn}}$ functionality which acts as an ideal synchronous delivery system. Using this method, parties explicitly send and receive all their (synchronized) messages by command requests to $\mathcal{F}_{\text{Syn}}$. In [78] the authors found an error in the original description of $\mathcal{F}_{\text{Syn}}$, later fixed again by Canetti as described in the current full version of [35]. On the constructive side, [78] proposes to give parties access to a global clock functionality, $\mathcal{F}_{\text{Clock}}$, instead. This functionality can be queried any time by any party in order to learn which is the present round. Assuming some predictability in the pattern of activation of parties (ITMs) in the protocol, then $\mathcal{F}_{\text{Clock}}$ is enough for synchronous communication and parties can handle the message delivery themselves, in contrast with $\mathcal{F}_{\text{Syn}}$.

Whether $\mathcal{F}_{\text{Syn}}$ [35] or $\mathcal{F}_{\text{Clock}}$ [78] is used, both functionalities keep track of a round number. This value is increased when, and only when, all honest parties indicate that they are done with their corresponding actions in a given round. This captures the issue of *rushing adversaries*, who wait to receive all honest messages in a given round before choosing, adaptively, what is going to be sent by malicious parties in that same round. As this is a possible behaviour in the real world, so it has to be in the ideal world.

Even though in the most strict formality we should use a functionality to model synchronous networks, this detail is skipped in the proofs presented in this thesis. We do so mostly to simplify their readability, as the intuitive notion of such communication model is not hard to grasp. We firmly believe that this is positive in terms of easing the job of reviewing the work here included. This is likely also the reason why an overwhelming majority of the MPC literature takes the same approach.

Finally and in conclusion to the presentation of the UC model, we can state the goal of the upcoming Chapters 3 and 4 to be that of securely realising the functionality $\mathcal{F}_{\text{MPC}}$ given in Figure 2.1. Chapter 5 has a very similar goal, the only difference with Figure 2.1 being that, as the protocols are designed to deal only with passive adversaries, the **Wait** command should be removed from it.

2.5 Garbled Circuits: The Two-Party Setting

Garbled circuits are the core behind all practical, constant round protocols for secure computation. Intuitively, they are a suitably encrypted version of the circuit's gates and wires that allows for its evaluation on a concrete set of inputs. Garbled circuit-based protocols have recently become much more efficient, and currently give the most practical approach for actively secure computation of binary circuits [107, 118] in the two-party scenario.

In this section we will focus on describing a particular variant of Yao's garbled circuits [132]. This will serve as an introduction to the more complex situation where more than two parties are involved in the protocol, which we will describe in Section 2.6.

2.5.1 Garbling in the Two-Party Setting

In Yao's protocol one party, the Garbler, constructs a garbled version of a mutually agreed Boolean circuit C while the other party, the Evaluator, evaluates it on an 'encrypted' version of both parties' inputs to get the final result. Stated more formally, the garbling process, which is computed entirely by the Garbler, goes as follows:

1. For each wire $w \in C$, sample a random wire mask $\lambda_w \in \{0,1\}$. Wire masks are used to encrypt the actual value x_w on the wire. We will call the value $\Lambda_w = x_w \oplus \lambda_w$ either the *colour bit* or the *external* or *public* value. We colour x_w either in red or blue to represent its associated colour bit Λ_w. More precisely, if $\lambda_w = 0$ then $x_w \in \{0,1\}$, whereas if $\lambda_w = 1$ then $x_w \in \{0,1\}$.

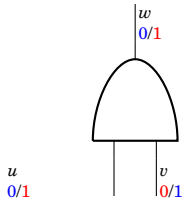
2. For each wire $w \in C$, two random keys $k_{w,\bullet}$, $k_{w,\bullet} \in \{0,1\}^\kappa$ are sampled.

3. Finally, let Enc be a Double Encryption scheme [91]. The garbled version $\tilde{g}$ of every $g \in C$ consists of the four following ciphertexts, which we call *rows*:

$$\tilde{g}_{\Lambda_u,\Lambda_v} = \mathsf{Enc}_{k_{u,\Lambda_u},k_{v,\Lambda_v}}(\Lambda_w || k_{w,\Lambda_w}), \quad (\Lambda_u,\Lambda_v) \in \{\bullet,\bullet\}^2$$

   where $\Lambda_w = f_g(\lambda_u \oplus \Lambda_u, \lambda_v \oplus \Lambda_v) \oplus \lambda_w \in \{\bullet,\bullet\}$ and f_g is the function computed by g, i.e., $f_g(x,y) = x \cdot y$ if g is an AND gate, or $f_g(x,y) = x \oplus y$ if g is a XOR gate.

As mentioned before, this is just one possible way of implementing Yao's garbled circuits, following in particular the point-and-permute paradigm [18, 97]. A visual description of the



Mask truth table $\longrightarrow$

u	v	w
0	0	0
0	1	0
1	0	0
1	1	1

Point & Permute $\longrightarrow$

Input	Garbled row
$(\bullet,\bullet)$	$Enc_{k_{u,\bullet},k_{v,\bullet}}(\bullet \| k_{w,\bullet})$
$(\bullet,\bullet)$	$Enc_{k_{u,\bullet},k_{v,\bullet}}(\bullet \| k_{w,\bullet})$
$(\bullet,\bullet)$	$Enc_{k_{u,\bullet},k_{v,\bullet}}(\bullet \| k_{w,\bullet})$
$(\bullet,\bullet)$	$Enc_{k_{u,\bullet},k_{v,\bullet}}(\bullet \| k_{w,\bullet})$

Figure 2.2: Point-and-permute garbling for an AND gate. Wire masks: $\lambda_u = 1, \lambda_v = 0, \lambda_w = 1$.

garbling process for an AND gate can be found in Figure 2.2, which we describe in more detail below.

Denote the left (resp. right) input wire u (resp. v) and let its wire mask be $\lambda_u = 1$ (resp. $\lambda_v = 0$). Consider now the first entry of an AND's gate truth table, i.e. $0 \wedge 0 = 0$. By applying the wire masks, we obtain corresponding external values $(\Lambda_u, \Lambda_v) = (1,0)$ which in the colour notation are $(\bullet,\bullet)$ and thus match the third row of the garbled gate. As the wire mask on the output wire is $\lambda_w = 1$, the plaintext to encrypt in that entry would then be $\bullet \| k_{w,\bullet}$. The remaining first, second and fourth entries of the garbled gate can be derived by the Garbler in the same way.

One of the most celebrated optimizations in garbled circuits is the Free-XOR technique [86], where the garbler samples a global random string $\Delta \in \{0,1\}^\kappa$ and the garbling algorithm is modified so that all the wire keys sampled on Step 2 of the garbling algorithm above are correlated:

$$k_{w,\bullet} \oplus k_{w,\bullet} = \Delta, \quad \forall w \in C$$

By introducing this correlation there is no more need to produce ciphertexts when garbling XOR gates, which can further be evaluated 'for free' just by XORing the keys and the external values of their input wires, i.e. $\Lambda_w = \Lambda_u \oplus \Lambda_v$ and $k_{w,\Lambda_w} = k_{u,\Lambda_u} \oplus k_{v,\Lambda_v}$.

2.5.2 Example Garbled Circuit

Before turning to the whole protocol description, we give an example garbled circuit in Figure 2.3. The function to compute is $f(w_1,w_2,w_3,w_4) = (w_1 \wedge w_2) \oplus (w_3 \vee w_4)$, where in our case the Garbler has inputs w_1, w_3 and the Evaluator has inputs w_2, w_4. In this example, the wire masks are:

$$\lambda_{w_1} = \lambda_{w_3} = \lambda_{w_4} = \lambda_{w_5} = \lambda_{w_7} = 1, \quad \lambda_{w_2} = \lambda_{w_6} = 0$$

If we look at them, the wire masks related to the AND gate $w_1 \wedge w_2 = w_5$ are exactly the same as those in Figure 2.2, for which a description of the garbling has already been provided. As the same procedure applies for the rest of the gates in the circuit, we proceed to show how a garbled circuit is evaluated. Assume that the Evaluator has obtained in some way the colour bits associated with the actual inputs of the circuit, as well as the keys corresponding to those.

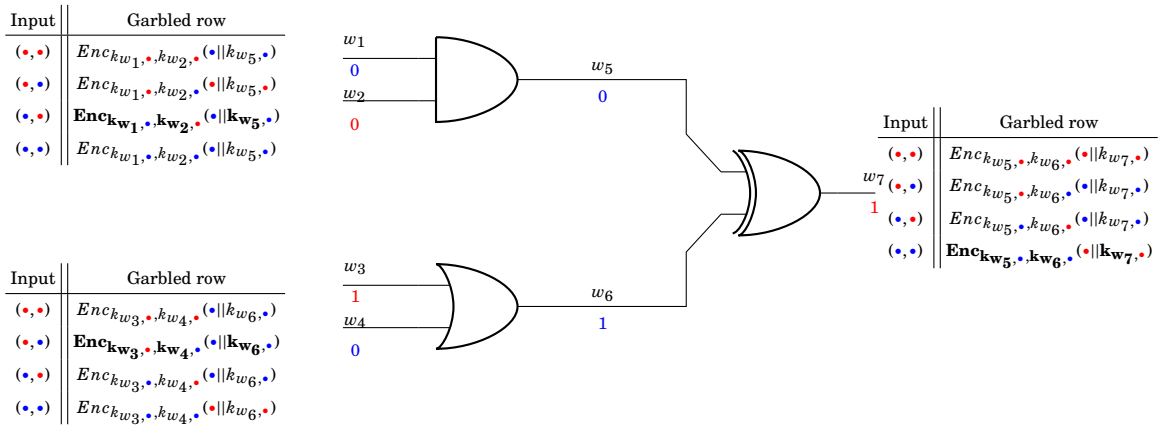


Figure 2.3: Evaluation of a garbled circuit. We mark in bold the rows that are decrypted by the parties when evaluating the circuit on the inputs displayed under the wires.

In Figure 2.3, for actual inputs $x_{w_1} = 0, x_{w_3} = 1$ from the Garbler, their corresponding colour bits are respectively $\bullet, \bullet$ and for inputs $x_{w_2} = 0, x_{w_4} = 0$ from the Evaluator, they translate to $\bullet, \bullet$. Therefore, their associated keys are $k_{w_1,\bullet}, k_{w_2,\bullet}, k_{w_3,\bullet}, k_{w_4,\bullet}$. We recall that the Evaluator does not know the value of the wire masks.

For the AND gate, the Evaluator has inputs $(\bullet, \bullet)$, so he has to decrypt that row of the garbled gate using keys $k_{w_1,\bullet}, k_{w_2,\bullet}$. By doing so, he learns that the colour bit on wire w_5 is $\bullet$, for which the key $k_{w_5,\bullet}$ relates. Analogously, using the colour bits and keys he has for wires w_3, w_4, he obtains colour bit $\bullet$ and key $k_{w_6,\bullet}$ for wire w_6. Finally, using the newly obtained keys, he can decrypt entry $(\bullet, \bullet)$ of the XOR gate and learn that the output wire colour is $\bullet$. Using the wire mask $\lambda_{w_7} = 1$ from the Garbler, this can be translated to an actual value of 1, which as expected corresponds to $f(0,0,1,0)$.

If the Free-XOR technique had been applied to the garbling scheme, there would be no garbled rows for the final XOR gate, so the table on the right of the picture should be ignored. From the Evaluator's perspective, the only difference would be that in order to compute $w_7 = w_5 \oplus w_6$, he would simply have to compute $k_{w_7,\bullet} = k_{w_5,\bullet} \oplus k_{w_6,\bullet}$, where he could compute the output wire's colour bit by XORing those of w_5 and w_6 – blue for both of them, which encodes an external value of one and so $1 \oplus 1 = 0 = \bullet$.

2.5.3 The Two-Party Computation Protocol

We can finally fully describe how to perform passively secure two party computation using Yao's garbled circuits:

1. The Garbler and the Evaluator agree on the boolean circuit $C(x_G, x_E) = y$ that they want to

compute, where x_G, x_E are the Garbler's and the Evaluator's private inputs, respectively.

2. The Garbler transforms C into a *garbled* circuit $\tilde{C}$ as described above. She also garbles her input x_G using the masks of the wires where she provides input. The result is $\tilde{x}_G$, consisting of a colour bit and its associated key for each of those wires.

3. For every bit of his input x_E, the Evaluator uses 1-out-of-2 OT to receive from the Garbler its garbled version $\tilde{x}_E$, suitable for evaluating $\tilde{C}$.

4. The Garbler sends the garbled input $\tilde{C}$ and her encoded input $\tilde{x}_G$ to the Evaluator, so he can now evaluate $\tilde{C}$. She also sends him the wire masks λ_w for every output wire w of the circuit, so he can decode the output of the computation.

5. The Evaluator computes $\tilde{C}(\tilde{x}_G, \tilde{x}_E)$ and obtains the colour bits associated with all output wires. Using the wire masks he received by the Garbler on the previous step, he computes the actual output and sends it back to the Garbler.

The overall efficiency of the protocol – irrespective of the nature of the adversary – is mostly related with the size of the garbled circuit, as the encoding of the inputs on Step 3 can be efficiently computed using OT extension [74, 83]. This realization led to a series of works on reducing the number of garbled rows in a gate [103, 111, 133] as well as variants of the previously described Free-XOR technique [84, 86].

In the following section we describe how to extend garbled circuits to be used by an arbitrary number of parties. As we will see, this requires the garbling step to be computed *obliviously* and, hence, the metrics for efficiency become not so tied to the size of the circuit as in the 2PC setting. Apart from the new difficulty when garbling, this different goal for optimization adds further interest to the field of concretely efficient, constant-round multi-party computation.

2.6 Multi-Party Garbled Circuits: BMR

The main challenge for the use of garbled circuits in the multi-party setting, when compared to Yao's protocol (cf. Section 2.5), is that no single party can be trusted to *garble* the circuit. Thus, this preprocessing step has to be performed *jointly* and *obliviously* using another MPC protocol. If that was not the case and the Adversary knew all the randomness used to garble the circuit, then the corrupted parties would be able to break security as long as one of them was also evaluating the circuit. In more detail, this arises from the fact that the XOR of the wire masks (which would be known by the Garbler) and the external values (known by the Evaluator) reveal the actual values on every wire in the circuit, as we described in Section 2.5.1.

Facing such a situation, one could think of implementing garbled circuits in the multi-party case by emulating the figure of the Garbler using an MPC protocol and then picking one party – or all of them! – to be the Evaluator. This is, in fact, the main idea behind the garbling technique

of Beaver, Micali and Rogaway (BMR) introduced in 1990 [18]. While such a method may initially sound as highly theoretical, which it was at the time, it has now become the core idea behind the most practical protocols based on garbled circuits, even in the two party setting [128].

The other fundamental observation in BMR is that all gates can be garbled *in parallel*. Hence, the depth of the circuit *computing the garbling* of a boolean circuit C equals that of the deepest circuit required to garble any single gate of C, denote that number by d. Recall that this 'garbling circuit' is the one we have to evaluate in order to obliviously distribute the figure of the Garbler. As most practical MPC protocols have a round complexity that depends linearly on the circuit depth, and this 'gabling' circuit has constant depth d, the garbling step becomes constant-round. This, when put together with the fact that the garbled circuit can also be evaluated in a constant number of rounds (as we will describe in Section 2.6.1) leads to the conclusion that BMR is a constant-round MPC protocol. This was the main contribution of BMR at a time where the only known mechanisms for constant-round MPC required an exponential blow-up in communication [14, 18].

Let's now look at the BMR construction in more detail. Similarly to Yao's garbled circuits (Section 2.5) denote by f_g be the function computed by each gate $g \in C$. The garbled version $\tilde{g}$ of that gate is defined in BMR as:

$$\tilde{g}_{\Lambda_u,\Lambda_v} = \mathsf{Enc}_{\boldsymbol{k}_{u,\Lambda_u},\boldsymbol{k}_{v,\Lambda_v}}(\boldsymbol{k}_{w,\Lambda_w}), \quad (\Lambda_u,\Lambda_v) \in \{0,1\}^2 \tag{2.1}$$

where $\Lambda_w = f_g(\lambda_u \oplus \Lambda_u, \lambda_v \oplus \Lambda_v) \oplus \lambda_w$ and $\mathsf{Enc}_{\boldsymbol{k}_{u,\Lambda_u},\boldsymbol{k}_{v,\Lambda_v}}$ is a 'vector double encryption' scheme that operates using two vectors of keys $\boldsymbol{k}_{u,\Lambda_u} = (k^1_{u,\Lambda_u},\ldots,k^n_{u,\Lambda_u}), \boldsymbol{k}_{v,\Lambda_v} = (k^1_{v,\Lambda_v},\ldots,k^n_{v,\Lambda_v})$ and can be instantiated under different assumptions. In Chapter 3 we implement it using PRFs, in Chapter 4 we use circular 2-correlation robust PRFs to enable Free-XOR, and in Chapter 5 we rely on a Syndrome Decoding problem in order for parties to use shorter encryption keys.

In the garbling phase, and for every wire $u \in C$, wire masks $\lambda_u \in \{0,1\}$ are randomly chosen and unknown to all parties. For wire keys, on the other hand and for each $b \in \{0,1\}$, every P_j knows a key $k^j_{u,b}$ from the vector of keys $\boldsymbol{k}_{u,b}$ associated with that wire. Importantly, these two statements imply that the vector of keys $\boldsymbol{k}_{w,\Lambda_w}$ encrypted on any given row (Λ_u,Λ_v) is unknown to all parties. This is due to the fact that MPC is used to compute $\Lambda_w = f_g(\lambda_u \oplus \Lambda_u, \lambda_v \oplus \Lambda_v) \oplus \lambda_w$ while keeping the wire masks secret, and Λ_w chooses between $\boldsymbol{k}_{w,0}$ and $\boldsymbol{k}_{w,1}$.

Whereas in Yao's garbled circuits the *size* of the garbled circuit was the metric most representative of efficiency, that turns out to be secondary now that garbling has to be performed using MPC. The dominant cost, hence, becomes the number of oblivious operations and their cost on the underlying protocol used to implement them. This opens up an interesting area of research for increasingly efficient constant-round MPC, where both the protocol used for garbling and the multi-party 'double encryption' scheme $\mathsf{Enc}_{\boldsymbol{k}_{u,\Lambda_u},\boldsymbol{k}_{v,\Lambda_v}}$ that it has to evaluate have to be adapted to each other. A time line for this area of work can be found in Section 2.6.2, but before that and for completeness, we will now turn to describing the evaluation phase of BMR.

Protocol	Adversary	Free-XOR	Other Remarks
BMR [18]	Active	✗	Honest majority, error in the proof.
TX03 [126]	Active	✗	Honest majority, splitter gates.
FairplayMP [23]	Passive	✗	Implementation paper.
BMR-SPDZ [93]	Active	✗	-
BMR-SHE [96] (§3)	**Active**	✗	**-**
BLO16 [25]	Passive	✓	-
BMR-OT [70] (§4)	**Active**	✓	**-**
Authenticated Garbling [129]	Active	✓	-
BLO17 [26]	Passive	✗	Constant gate size.
Arithmetic Garbling [24]	Passive	(✓)	Honest majority, arithmetic circuits.
BO17 [27]	Pass/Act	✓	Honest majority.
TinyKeys [69] (§5)	**Passive**	✗	**FleXOR, splitter gates.**

Table 2.1: Comparison of efficient protocols based on garbled circuits which can support any number of parties. Works are listed in chronological order.

2.6.1 Evaluating BMR Garbled Circuits

Given the description of the garbling phase above, the evaluation phase of BMR is almost identical to the one described in Section 2.5. Only two more conditions are missing from the description of the *preprocessing* protocol in the previous section:

1. Wire masks for the circuit output wires have been revealed to all parties.

2. For every input wire in the circuit, its corresponding wire mask has been revealed to the party providing input on it, and only to that party.

Taking the previous points into account, another advantage of BMR is that each party can now play the role of the Evaluator. This saves the last round in Yao, where the Evaluator communicates the output to the Garbler. Evaluation goes as follows: First, parties broadcast the colour bits associated with their respective actual inputs to the circuit. Once this is done, everyone broadcasts their keys corresponding to those wires and colour bits. As parties hold now the vectors of keys corresponding to input wires, they can decrypt the relevant row in each garbled gate attached to them. Upon completion, the parties compute the actual output using the output wire masks.

2.6.2 A Brief History of Efficient Multi-Party Garbled Circuits

The use of garbled circuits for secure computation between an arbitrary number of parties was first introduced by Beaver, Micali and Rogaway in 1990 [18]. Their paper, whereas foundational, presented an insecure construction. This flaw in their result was not discovered until 2003, when

Tate and Xu [126] remarked that the 'vector double encryption' used in [18] is not secure when multiple gates share a common input wire. The error can be easily described: In order to garble the gates, wire 'keys' were used as seeds for a pseudo-random generator, so encryption looked like:

$$\mathsf{Enc}_{\boldsymbol{k}_{u,\Lambda_u},\boldsymbol{k}_{v,\Lambda_v}}(\boldsymbol{k}_{w,\Lambda_w}) = \boldsymbol{k}_{w,\Lambda_w} + \sum_{i=1}^{n} PRG(k^i_{u,\Lambda_u}) + PRG(k^i_{v,\Lambda_v})$$

Hence, when multiple plaintexts are 'encrypted' under one of the two key vectors, the resulting 'ciphertext' does not use independent pseudorandom masks. Such correlation is not only a theoretical concern, as it can be exploited even by passive adversaries to break the security of the protocol. A good example of the attack can be found in [126]. The solution proposed by Tate and Xu was the introduction of *splitter gates*, which take as input one wire and output two new ones, carrying the same value. By placing trees of splitter gates in the original circuit to garble, the resulting one can be made such that no wire is ever input to two different gates. This, plus a method for garbling splitter gates, fixes the issue in [18]. Even though their proposal works, it is frequently not the most efficient solution, with notable exceptions [69]. Almost every other protocol building on BMR solves the same problem by using pseudo-random *functions* instead of pseudo-random *generators*. By adding a unique identifier to each gate which is then used as an input to the PRF, independent pseudo-random masks can be produced.

While two-party computation based on Yao's garbled circuits started booming [1, 57, 88, 111, 124] after its first full-fledged implementation in the 2004 Fairplay paper [97], BMR remained almost forgotten, with two exceptions: The FairplayMP implementation in 2008 [23] and, seven years after that, the BMR-SPDZ protocol by Lindell et al [93]. Apart from bringing the community's attention back to the problem, one of their biggest achievements was that of getting rid of the zero-knowledge proofs required in [18] to ensure that parties provide the right evaluations of PRGs/PRFs when garbling the gates. The efficiency of their approach was then improved in [96], which constitutes Chapter 3 in this thesis.

Many techniques used in two-party garbled circuits do not translate well to BMR. Improvements reducing the circuit size such as row reduction [103, 111, 133] turn out to be counterproductive when all parties have to obliviously garble the circuit together. Nevertheless, the highly celebrated Free-XOR technique [86] can also be easily applied to BMR, as it was first noticed in [25]. Although the authors only provide a passively secure protocol, their paper also includes interesting experimental results where BMR is confirmed to be faster than secret-sharing based protocols for deep circuits in slow networks.

Later on, Free-XOR was also extended to the active setting in concurrent works [70, 129]. Both show how to leverage information-theoretic MACs from the TinyOT protocol [58, 104] to act as Free-XOR correlations, which makes TinyOT a natural choice for BMR garbling. Additionally, [70] provides an efficient and general transformation for non-constant round protocols to Free-

XOR-enabled BMR. More differences between both works exists, which are further described during the presentation of [70] in Chapter 4.

A common feature of all the BMR constructions above is that garbled gates have a size of around $4 \cdot n \cdot \kappa$ bits, where n is the number of parties and κ the security parameter. This is due to the fact that each row in the garbled gate consists, basically, of one ciphertext per party. At the time of writing, two works look at how to reduce this size, which can be particularly problematic for large-scale scenarios. As it is not an easy task, they both deal with passive adversaries only. The first one is [26], which uses key-homomorphic PRFs to add all the n ciphertexts in a row together, obtaining garbled gates whose size does not depend in the number of parties. TinyKeys [69], presented in Chapter 5, takes a different approach from the heavy public key machinery of [26]. The solution we present involves making the individual keys for the 'vector double encryption' of a small size $\ell \ll \kappa$ and make security rely on the concatenation of all honest parties' short keys. This reduces not only communication, but also the computational cost of evaluating each gate when compared with [25].

Finally, other works in the area include [27], which looks at efficiency improvements for BMR in the honest majority setting and [24], which garbles arithmetic circuits rather than boolean ones. A chronological synthesis of our overview can be found in Table 2.1.

2.7 A Note on Actively Secure Garbled Circuits

The exposition of both Yao's and Beaver, Micali and Rogaway's protocols in the previous sections are limited to security against passive adversaries. Whereas this is a relevant setting and a good starting point when figuring out the core ideas and limitations of MPC protocols, we ideally want security to hold against active adversaries too. In this section we review some of the methods used to achieve this more difficult goal

One of the most explored directions for enabling active security in Yao's protocol is the cut-and-choose technique. Simplifying, this consists in the Garbler constructing many garbled circuits for the commonly agreed function and then sending all of them to the Evaluator. Next, the Evaluator asks the Garbler to reveal the randomness used to produce e.g. half of the circuits, in order to check that they were constructed correctly. If that is the case, the Evaluator can then evaluate the other half and obtain the result. Otherwise, the Evaluator has detected malicious behaviour from the Garbler and aborts.

More subtleties are part of cut-and-choose and the technique has evolved through a long line of work that applies it either at the circuit level (as described in [1, 57, 88, 90, 92, 124]), the gate level [105, 108, 134] or sub-circuits in between [85]. While such method notoriously improved the efficiency of actively secure 2PC in the last decade, it has been implicitly proved as unattractive for BMR so far. This is due to the fact that, as garbling has to be computed obliviously, there are more efficient techniques for ensuring active security of the underlying MPC gadgets constructing

the garbled circuit.

Another very efficient approach for active security in 2PC relies on the *dual execution* paradigm [99, 100, 118]. In this framework, parties run two instances of Yao's protocol, taking a different role (amongst Garbler and Evaluator) in each one. Once they have done this, they securely test for equality the (garbled) outputs received by the Evaluator in each instance, and reveal them if the test passes. Intuitively, this then means that malicious party computed the garbled circuit correctly. Nevertheless, the equality test step leaks information on the honest parties' input and in order to achieve full security, cut-and-choose techniques are required once again. Dual execution could be interpreted to have been extended to the multi-party setting in [66], but their solution only works for a very concrete scenario in terms of number of parties and corruption threshold.

The conclusion to all the above is that obtaining efficient, actively secure protocols for BMR is not an easy task and requires to think differently. An evolution of that process of thought can be observed by reading the papers referenced in Section 2.6.2 or, more accessibly, by reading the two following chapters 3 and 4, which are also presented in chronological order.

CHAPTER 3

GARBLING USING SOMEWHAT HOMOMORPHIC ENCRYPTION

This chapter is based on joint work with Yehuda Lindell and Nigel Smart [96], which was presented at TCC 2016-B.

In this chapter we introduce two similar constant round multi-party protocols secure against an active adversary corrupting up to $n-1$ out of a total n parties. The work we present sits philosophically between Gentry's MPC protocol using Fully Homomorphic Encryption [60] and the best previous protocol for generating BMR circuits against the same kind of adversary [93], which we will dub BMR-SPDZ. Our protocols avoid various inefficiencies of those two results. Compared to Gentry's protocol we only require Somewhat Homomorphic Encryption (SHE) and moreover we provide security against malicious adversaries, which [60] does not. In comparison to BMR-SPDZ we require only a quadratic communication complexity in the number of players (as opposed to cubic), we have fewer rounds and we require fewer proofs of correctness of ciphertexts.

Our two new protocols follow the BMR paradigm described in Section 2.6 and use homomorphic encryption in order to garble the boolean circuit describing the computation. The main difference between both is how the circuit representing the *garbling* of the boolean gates is described. The SHE scheme has to support plaintext spaces of $\mathbb{F}_p$, where $p > 2^\kappa$. Our first, conceptually simpler protocol is described in Section 3.4, whereas the second one (given in Section 3.5) allows to perform garbling with a lower depth arithmetic circuit and requires more multiplications in both the garbling and evaluation phases of BMR.

3.1 Introduction

The work presented in this chapter focuses on the construction of concretely efficient MPC protocols that run in a constant number of rounds. The approach we follow consists of constructing

a two phase protocol. In the first phase the parties use a generic MPC protocol to construct a garbled circuit for the function being computed. Then, in a constant round evaluation phase the garbled function is evaluated. The first garbling phase works in a gate-by-gate manner, and so by processing all gates in parallel we obtain a circuit of constant depth which can be evaluated in a constant number of rounds via any MPC protocol.

In [93] an efficient variant of the BMR protocol is used which utilizes the SPDZ [47] generic MPC protocol in the first garbling phase. In addition, the authors introduce other optimizations which make the entire protocol actively secure for very little additional overhead. The SPDZ protocol itself uses a two phase approach, in the first phase, which utilizes Somewhat Homomorphic Encryption, correlated randomness is produced. Then in the second phase this correlated randomness is used to evaluate the desired functionality (which in this case is the BMR garbling). Thus overall this protocol, which we dub BMR-SPDZ, consists of three phases: A phase using SHE, a phase doing generic MPC via the SPDZ online phase, and the final BMR circuit evaluation phase.

There is another approach to constant round MPC, which utilizes Fully Homomorphic Encryption (FHE) [61]. FHE is similar to the notion of public key encryption, with the added property that given any set of FHE ciphertexts $c_1 = \mathsf{Enc}(m_1), \ldots, c_k = \mathsf{Enc}(m_k)$ it allows to compute without interaction a ciphertext c such that $\mathsf{Dec}(c) = f(m_1, \ldots, m_k)$. The function f can be any circuit, and the production of c is referred to as the *homomorphic evaluation* of f on the original ciphertexts. If the circuits f that are allowed to be evaluated are restricted to a certain maximum *depth* we obtain the notion of Somewhat Homomorphic Encryption (SHE), where much more efficient constructions can be obtained.

In Gentry's FHE-based MPC protocol [60], parties simply input their data using the encryption of the underlying FHE scheme and then evaluate the function. Finally, in order to obtain the result, they perform a distributed decryption (which requires $R_{\mathsf{Out}} = 2$ rounds of interaction with current FHE schemes). This protocol is essentially optimal in terms of the number of rounds of communication, but it suffers from a number of drawbacks. The major drawback is that it requires FHE, which is itself a prohibitively expensive operation (and currently not practical). In addition, it is not immediately clear how to make the protocol actively secure without incurring significant additional costs. Here we outline how to address this latter problem, as a by-product of the analysis of our main protocol.

3.1.1 Our Contributions

Returning to the BMR based approach we note that *any* MPC protocol could be used for the BMR garbling phase, as long as it can be made actively secure within the specific context of the BMR protocol. In particular we could utilize Gentry's FHE-based MPC protocol (using only a SHE scheme) to perform the first stage of the BMR protocol, a protocol idea which we shall denote by BMR-SHE. The main observation as to why this is possible is that, as we have mentioned, the

depth of the circuit computing the BMR garbled circuit is itself constant and equal to the depth of the circuit required to compute a single garble gate. We therefore conclude that *somewhat* homomorphic encryption suffices.

A number of problems arise with this idea, which we will address in the following. First, can we make the resulting protocol actively secure for little additional cost? Second, is the required depth of the SHE scheme sufficiently small to make the scheme somewhat practical? Recall the BMR-SPDZ protocol only requires the underlying SHE scheme to support circuits of multiplicative depth one, and increasing the depth increases the cost of the SHE itself. Third, is the resulting round complexity of the scheme significantly less than that of the BMR-SPDZ protocol? Note that we can only expect a constant factor improvement, but such constants matter in practice. Fourth, can we save on any additional costs of the BMR-SPDZ protocol?

Since we use Gentry's FHE-based protocol – more precisely, an SHE version of it – we now outline two of its key challenges, which also apply to our protocol. When entering data we require an actively secure protocol to encrypt the FHE data, in particular we need to guarantee to the receiving parties that each encryption is well formed. The standard technique to do this is to also transmit a zero-knowledge proof of the correctness of encryption. A method to do this can be found in [44, Appendix F] or [15, Section 3.2]. This is costly, and in practice rather inefficient. We call this protocol ID, and the associated round cost by R_{ID}. In addition if we need to make further *input dependent inputs*, then this round cost will multiply. Thus we also need to introduce a sub-protocol with round cost $R_{\mathsf{Input+}} = 1$, which enables us to place all the zero-knowledge proofs for proving correctness of input into a preprocessing phase.

The second problem with Gentry's protocol is that we need to ensure, in a concretely efficient way, that the distributed decryption is also actively secure. We present a sub-protocol $\mathsf{Out+}$ performing this task, which has an associated round complexity of $R_{\mathsf{Out+}} = 2$ rounds. To the best of our knowledge such solution did not exist in the literature at the time of writing, but a similar, less efficient solution based on algebraic manipulation detection codes was introduced concurrently in [48].

By utilizing the actively secure input and output routines in Gentry's protocol we obtain an actively secure variant of Gentry's FHE based protocol which we denote by G_a. This is in addition to the original passively secure FHE based protocol of Gentry which we denote by G_p.

In summary, we actually obtain two distinct protocols that we describe below:

DEPTH 4 PROTOCOL $\Pi_{\mathrm{Depth}-4}$ (SECTION 3.4). In some sense we can think of our basic BMR-SHE protocol as the same as the BMR-SPDZ protocol of [93], but cutting out the need of producing multiplication triples and the interaction needed to evaluate the garbling via the online phase of SPDZ. Indeed almost all of our basic protocol is identical to that described in [93]. However, naively applying SHE to the protocol from [93] results in a protocol that is neither efficient nor secure. For example, naively applying Gentry's MPC protocol to the garbling stage would result in needing an SHE scheme which supports a depth logarithmic

in the number of parties n, whereas we would rather utilize a SHE scheme with constant depth. Thus we need to carefully design the FHE based MPC protocol to realise the BMR garbled circuit.

DEPTH 3 VARIANT $\Pi_{\text{Depth}-3}$ (SECTION 3.5). We present a variant of our protocol which reduces the depth of the required SHE scheme, at the expense of requiring each party to input a larger amount of data. Interestingly, the main aim of the design in the BMR-SPDZ protocol was to reduce the number of multiplications needed (since each multiplication required generating a multiplication tuple for SPDZ, and this is its main cost). In contrast, when using SHE directly, additional multiplications are not so expensive as long as they are carried out in parallel. Stated differently, the main concern is the *depth* of the circuit computing the BMR garbled circuit, and not necessarily its size. Of course, for concrete efficiency, one must try to minimize both, as reducing one slightly while greatly increasing the other would not be beneficial. In order to achieve this reduction in the depth of the circuit computing the BMR circuit, we utilize an observation that when computing the garbled circuit it suffices to obtain either the PRF key on the output wire or its additive inverse. This is due to the fact that we can actually take the PRF key to be the *square* of the value obtained in the garbled gate, which is the same whether k or $-k$ is obtained[1]. This allows us to combine the generation of the external values and their corresponding vector of keys together. The additional flexibility of being able to output either the key or its additive inverse allows us to reduce the required SHE depth from *four* to *three*.

3.1.2 Comparison

By way of comparison we outline in Table 3.1 the differences between our two protocol variants and those of Gentry and BMR-SPDZ. We let n denote the number of parties, W and G denote the number of wires and gates in the binary circuit respectively, and W_{in} the number of input wires. To ease counting of rounds we consider a secure broadcast to be a single round operation (in the case of a dishonest majority, where parties may abort, a simple two-round echo-broadcast protocol suffices in any case [65]). We will see later that $R_{\mathsf{Out+}} = 2$, and $R_{\mathsf{ID}} = 3$. In the table the various functions T_1, T_2, T_3 describing the number of executions of ID are

$$
\begin{aligned}
T_1 &= 16 \cdot G \cdot n^3 + (8 \cdot G + 4 \cdot W) \cdot n^2 + 9 \cdot W \cdot n + 156 \cdot G \cdot n, \\
T_2 &= 4 \cdot G \cdot n^2 + (3 \cdot W + 1) \cdot n, \\
T_3 &= (4 \cdot G + 2 \cdot W) \cdot n^2 + (W + 1) \cdot n.
\end{aligned}
$$

If we compare the BMR-SPDZ protocol with our protocol variants $\Pi_{\text{Depth}-4}$ and $\Pi_{\text{Depth}-3}$ we see that the major difference in computational cost is the number of invocations of the protocol ID.

[1] Note that the resulting PRF key has one bit of entropy less than k, which can be compensated by longer plaintext spaces in the SHE scheme.

Protocol	Security	**Rounds of Interaction**	**Depth of FHE/SHE**	**Number of ID Execs**
G_p	passive	$3 = 1 + R_{\mathsf{Out}}$	Depth of f	0
G_a	active	$5 = R_{\mathsf{ID}} + R_{\mathsf{Out+}}$	$1 +$ Depth of f	$n + W_{in}$
BMR-SPDZ	active	$16 = 13 + R_{\mathsf{ID}}$	1	T_1
$\Pi_{\text{Depth}-4}$	active	$9 = R_{\mathsf{ID}} + 4 + R_{\mathsf{Out+}}$	4	T_2
$\Pi_{\text{Depth}-3}$	active	$9 = R_{\mathsf{ID}} + 4 + R_{\mathsf{Out+}}$	3	T_3

Table 3.1: Comparison of Gentry's, the BMR-SPDZ and our protocol.

The difference between BMR-SPDZ and $\Pi_{\text{Depth}-4}$ is equal to $T_1 - T_2 = 16 \cdot G \cdot n^3 + 4 \cdot (G + W) \cdot n^2 + (6 \cdot W - 1) \cdot n + 156 \cdot G \cdot n$ invocations. To be very concrete, for 9 parties, a circuit of size 10,000 gates and wires, the number of ID invocations equals 141,210,000 in BMR-SPDZ versus 3,510,009 in BMR-SHE-$\Pi_{\text{Depth}-4}$ versus 4,950,009 in BMR-SHE-$\Pi_{\text{Depth}-3}$. Thus, $\Pi_{\text{Depth}-4}$ is *one fortieth* of the cost of BMR-SPDZ, and $\Pi_{\text{Depth}-3}$ is *one twenty-eighth* of the cost of BMR-SPDZ. This gap widens further as the number of parties grows, with the difference for 25 parties being a factor of 100 for $\Pi_{\text{Depth}-4}$ and 70 for $\Pi_{\text{Depth}-3}$. We remark, however, that even for just 3 parties, protocols $\Pi_{\text{Depth}-4}$ and $\Pi_{\text{Depth}-3}$ are already one twenty-third and one eighteenth of the cost, respectively.

On the downside we require an SHE scheme which will support depth three or four circuits, as opposed to the depth one circuits of the BMR-SPDZ protocol. The SHE scheme needs to support message spaces of $\mathbb{F}_p$, where $p > 2^\kappa$. We use [42], which gives potential parameter sizes for various SHE schemes supporting depth two and five, and run the experiments there to compare the parameters required for our specific depths here (depth 1 for SPDZ, depth 4 for protocol $\Pi_{\text{Depth}-4}$ and depth 3 for protocol $\Pi_{\text{Depth}-3}$). Specifically, assuming ciphertexts live in a ring R_q, then the dimension needs to go up by approximately a factor of 1.5 for depth-3 and a factor of 2 for depth-4, and the modulus by a factor of 1.6 for depth-3 and a factor of 2 for depth-4. Assuming standard DCRT representation of R_q elements, this equates to an increase in the ciphertext size by a factor of approximately 2.4 for depth-3, and by approximately a factor of 4 for depth-4. Furthermore, the performance penalty (cost of doing arithmetic) increases by a factor of approximately 3.6 for depth-3, and by a factor of 8 for depth-4. Factoring in this additional cost, we have that when compared to BMR-SPDZ, the relative improvement in the computational cost in the above example becomes a factor of 40/8 = 5 for $\Pi_{\text{Depth}-4}$ and 28/3.6 = 7.7 for $\Pi_{\text{Depth}-3}$ for 9 parties, and a factor of 100/8 = 12.5 for $\Pi_{\text{Depth}-4}$ and 70/3.6 $\approx$ 19.4 for $\Pi_{\text{Depth}-3}$ for 25 parties. Thus, both $\Pi_{\text{Depth}-4}$ and $\Pi_{\text{Depth}-3}$ significantly outperform BMR-SPDZ, and the depth reduction carried out in $\Pi_{\text{Depth}-3}$ provides additional speedup (and reduction in bandwidth).

3.1.3 Additional Related Work

Our work should also be compared to [41] in which a constant round 3PC protocol is given, based on Yao's garbled circuits. However, active security in their case is provided by an expensive

cut-and-choose protocol. In addition, their protocol is specifically designed for the three-party case, whereas we consider multiparty computation for any number of parties. Another constant-round multiparty protocol was constructed by [75]. However, although this protocol has good asymptotic complexity, its concrete efficiency is very unclear and no concretely efficient instantiation has been found. In [89] the authors propose a concrete instantiation of [75] for the two-party case, but no analogous proposal exists for the multiparty case.

3.2 Preliminaries

As a warm up to our protocols, we first give an outline of Gentry's FHE based protocol and to the previous work by Lindel et all. [93], which uses SPDZ to produce BMR garbled circuits.

3.2.1 A Basic FHE Functionality With Distributed Decryption

We first describe in Figure 3.1 a basic FHE functionality which contains a distributed decryption functionality. Two points need to be noted about the functionality: Firstly, the distributed decryption operation in **Output** can produce an incorrect result, but this is limited to an additive error which is introduced by the adversary before revealing the output. Secondly, the **InputData** routine is actively secure, and so a proof of correctness of its decryption is needed for each input ciphertext. The need for such an actively secure input routines is because we need to ensure that parties enter *valid* FHE/SHE encryptions, and that the simulator can *extract* the plaintext values. Within the functionality we denote the depth of a varable x by $D(x)$, and we describe how the depth is altered with each operation which can affect the depth.

A method to perform the required **InputData** operation is given in [44, Appendix F], or [15, Section 3.2]. The basic idea is to check a number of executions of **InputData** at the same time. The protocol runs in two phases, in the first phase a set of reference ciphertexts are produced and via cut-and-choose one subset is checked for correctness, while the other is permuted into buckets; one bucket for each value entered via **InputData**. In the second phase the input ciphertexts are checked for correctness by combining them homomorphically with the reference ciphertexts and opening the result. We denote the round complexity of the protocol implementing **InputData** by R_{ID}. An analysis of the protocol from [44] indicates that it requires $R_{\mathsf{ID}} = 3$ rounds of communication: In the first round of the proof one party broadcasts the reference ciphertexts, in the next round the parties choose which ciphertexts to open, and in the third round the ciphertexts are combined and opened.[2] Thus, overall, three rounds suffice.

In the following, we fix the notation $\langle varid \rangle$ to represent the result stored in the variable varid by the $\mathscr{F}_{\text{FHE}}/\mathscr{F}_{\text{SHE}}$ functionalities. In particular, we will use the arithmetic shorthands $\langle z \rangle = \langle x \rangle + \langle y \rangle$ and $\langle z \rangle = \langle x \rangle \cdot \langle y \rangle$ to represent the result of calling the **Add** and **Multiply** commands

[2] Choosing at random which ciphertexts to open cannot be carried out in a single round. However, it is possible for all parties to commit to the randomness in previous rounds and only decrypt in this round.

The FHE Functionality: $\mathcal{F}_{\text{FHE}}$/$\mathcal{F}_{\text{SHE}}$

The functionality consists of externally exposed commands **Initialize**, **InputData**, **Add**, **Multiply** and **Output**, and one internal subroutine **Wait**.

Initialize: On input (*init*,p) from all parties, the functionality activates and stores p. All additions and multiplications below will be mod p.

Wait: The functionality waits on the adversary for a reply. If $\mathcal{A}$ returns $\perp$ then the functionality aborts, otherwise it continues.

InputData: On input (*input*,P_i,varid,x) from P_i and (*input*,P_i,varid,?) from all other parties, with varid a fresh identifier, the functionality stores (varid,x). The functionality then calls `Wait`.

Add: On command (*add,* varid_1, varid_2, varid_3) from all parties (if varid_1, varid_2 are present in memory and varid_3 is not), the functionality retrieves (varid_1, x), (varid_2, y) and stores ($\text{varid}_3, x+y \mod p$).

Add-scalar: On command (*add-scalar, a,* varid_1, varid_2) from all parties (if varid_1 is present in memory and varid_2 is not), the functionality retrieves (varid_1, x) and stores ($\text{varid}_2, a+x \mod p$).

Multiply: On command (*multiply,* varid_1, varid_2, varid_3*) from all parties (if* varid_1, varid_2 are present in memory and varid_3 is not), the functionality retrieves (varid_1,x), (varid_2,y) and stores (varid_3,$x \cdot y$ mod p).

In the case of the $\mathcal{F}_{\text{SHE}}$ version of this functionality only a limited depth of such commands can be performed, which is specified for the functionality.
Depth Cost: $D(\text{varid}_3) = \max(D(\text{varid}_1), D(\text{varid}_2)) + 1$.

Multiply-scalar: On command (*multiply-scalar, a,* varid_1, varid_2) from all parties (if varid_1 is present in memory and varid_2 is not), the functionality retrieves (varid_1, x) and stores ($\text{varid}_2, a \cdot x \mod p$).

Output: On input (*output,*varid*, i*) from all honest parties (if varid is present in memory), and a value $e \in \mathbb{F}_p$ from the adversary, the functionality retrieves (varid,x), and if $i = 0$ it outputs (varid,x) to the adversary. The functionality then calls `Wait`. If `Wait` does not result in an abort, then the functionality outputs $x + e$ to all parties if $i = 0$, or it outputs $x + e$ only to party i if $i \neq 0$.

Figure 3.1: The FHE/SHE Functionality: $\mathcal{F}_{\text{FHE}}$/$\mathcal{F}_{\text{SHE}}$

in the $\mathcal{F}_{\text{FHE}}$/$\mathcal{F}_{\text{SHE}}$ functionality, and we will slightly abuse those shorthands to denote subsequent additions or multiplications.

The description of **Output** in the case of a passively secure functionality is identical to the behaviour of the standard distributed decryption procedure for FHE schemes such as BGV, again see [44] for how the distributed decryption is performed. The basic protocol is to commit to the distributed decryption shares, and then open the shares. This gives a round complexity for **Output** of $R_{\text{Out}} = 2$. We shall provide a simple mechanism to provide active security for the **Output** command in the next section, which comes at the expense of increasing the required supported depth of the SHE scheme by one.

In the case of a passively secure variant of the FHE functionality, one would always have $e = 0$ in the **Output** routine. Furthermore, we would not need a proof of correctness of the input ciphertexts and so the number of rounds of interaction in the **InputData** routine would be $R_{\text{ID}} = 1$.

3.2.2 Gentry's FHE-Based MPC Protcol

In [60] Gentry presents an MPC protocol which has optimal round complexity to implement $\mathscr{F}_{\text{MPC}}$. In the $\mathscr{F}_{\text{FHE}}$-hybrid model the protocol can be trivially described as follows: The parties enter their data using the **InputData** command of the FHE functionality, the required function is evaluated using the **Add** and **Multiply** commands (i.e. each party locally evaluates the function using the FHE operations). The **Add-scalar** and **Multiply-scalar** commands can be computed by the parties locally encrypting the scalar with a mutually agreed randomness (so that all hold the same ciphertext) and then using the regular FHE **Add** or **Multiply** command, respectively.

Finally, the output is obtained using the **Output** command of the FHE functionality. For passively secure adversaries this gives us an 'efficient' MPC protocol, assuming the FHE scheme can actually evaluate the function. For active adversaries we then have to impose complex zero-knowledge proofs to ensure that the **InputData** command is performed correctly, and we need a way of securing the **Output** command (which we will come to in Section 3.3).

3.2.3 The BMR-SPDZ Protocol

We shall now overview the BMR-SPDZ protocol from [93]. Much of the details we cover here focus on the offline SHE-part of the SPDZ protocol and how it is used in the BMR-SPDZ protocol. Recall the SPDZ protocol makes use of two phases: An offline phase which uses an SHE scheme (which for our purposes we model via the functionality $\mathscr{F}_{\text{SHE}}$ above restricted to functions of multiplicative depth one) and an online phase using (essentially) only information theoretic constructs. These two phases are used to create a shared garbled circuit which is then evaluated in a third phase in the BMR-SPDZ protocol.

First Phase Costs

The first phase of the BMR-SPDZ protocol requires an upper bound on the total number of parties n, internal wires W, and gates G of the circuit which will be evaluated. The phase then calls the offline phase of the SPDZ engine to produce $M = (4 \cdot n + 5) \cdot G$ multiplication triples, $B = W$ shared random bits, $R = 2 \cdot W \cdot n$ shared random values and $I = 8 \cdot G \cdot n$ shared values for entering data per party.

The main cost of the BMR-SPDZ protocol is actually in computing this initial data, yet the paper [93] does not address this cost in much detail. Delving into the paper [44] we see that each of these operations requires parties to encrypt random data under the SHE scheme and to produce additive sharings of SHE encrypted data. This first operation is identical to our *input* command on the functionality $\mathscr{F}_{\text{SHE}}$. We delve into the costs of the operations in more detail:

- **Encrypting (Input) Data** ID**:** When a party produces a ciphertext we need to ensure that it has a valid form, so as to protect against active attackers. As remarked above this is done using a zero-knowledge proof of correctness. Whilst the computational costs of this can be

amortized due to 'packing' in the SHE scheme, it is a non-trivial cost per encryption. We shall denote the computational and round cost in what follows by C_{ID} and R_{ID} respectively, i.e. the computational and round cost of the actively secure *EncCommit* operation from [44].

- **Producing Random Resharings:** Given a ciphertext encrypting a value m this procedure results in an additive sharing of m amongst the n parties. The computational cost of this procedure is dominated by the invocations of the ID protocol. Since each party needs to encrypt a random value, the computational cost $n \cdot C_{\mathsf{ID}}$ and the round complexity is $R_{\mathsf{ID}} + 1$. Again, the computational costs can be amortized due to the packing of the SHE scheme.

- **Producing Multiplication Triples:** To produce an unchecked triple this requires (per party) the encryption of two random values (of a_i and b_i in the triple $([a],[b],[c])$), plus four resharings (three of which can be done in parallel, with the fourth only partially in parallel). To produce a checked triple, this needs to be done twice (in parallel), followed by a sacrificing step of one of the triples via a procedure (described in [44]) which requires another two rounds of interaction. Thus the total computational cost is dominated by $12 \cdot n \cdot C_{\mathsf{ID}}$ and the round complexity is $R_{\mathsf{ID}} + 4$.

- **Producing Shared Random Bits:** To produce an unchecked random bit we require (per party) the encryption of one random value, one passively secure distributed decryption (with only one round of interaction), plus two resharings (in parallel). To produce a checked random bit, the above has to be combined with an unchecked multiplication triple in a sacrificing step which requires two rounds of interaction. Thus the total computational cost is dominated by $9 \cdot n \cdot C_{\mathsf{ID}}$ and the round complexity is $R_{\mathsf{ID}} + 4$.

- **Producing Shared Random Values:** This requires (per party) the encryption of one random value, and two resharings which can be done in parallel. Thus the total computational cost is $2 \cdot n \cdot C_{\mathsf{ID}}$, and the round complexity is $R_{\mathsf{ID}} + 1$.

- **Producing Input Data:** Per data item which needs to be input for each player this requires the encryption of one random value plus two resharings (which cannot be fully parallelised), as well as one additional round of interaction. Thus the total computational cost is dominated by $C_{\mathsf{ID}} + 2 \cdot n \cdot C_{\mathsf{ID}}$, and the round complexity is $R_{\mathsf{ID}} + 3$.

A major bottleneck in the protocol, which is also one of the requisites for active security, is the cost of encrypting the random data described above. If we combine the various formulae, we obtain a dominant cost of $T_1 \cdot C_{\mathsf{ID}}$, where T_1 is the number of calls to the ID protocol:

$$\begin{aligned} T_1 \cdot C_{\mathsf{ID}} &= 12 \cdot n \cdot C_{\mathsf{ID}} \cdot M + 9 \cdot n \cdot C_{\mathsf{ID}} \cdot B + 2 \cdot n \cdot C_{\mathsf{ID}} \cdot R + (1 + 2 \cdot n) \cdot n \cdot C_{\mathsf{ID}} \cdot I \\ &= (12 \cdot (4 \cdot n + 5) \cdot G + 9 \cdot W + 4 \cdot W \cdot n + (1 + 2 \cdot n) \cdot n \cdot 8 \cdot G) \cdot n \cdot C_{\mathsf{ID}} \\ &= (16 \cdot G \cdot n^3 + (56 \cdot G + 4 \cdot W) \cdot n^2 + 9 \cdot W \cdot n + 60 \cdot G \cdot n) \cdot C_{\mathsf{ID}} \end{aligned}$$

which is *cubic* in the number of players.

The total round complexity of the SPDZ offline phase is the maximum round complexity of the various preprocessing operations in the SPDZ offline phase, namely $R_{\mathsf{ID}} + 4$. This holds since the transmission of *all* random encrypted values can occur in one round at the beginning of this phase. We stress that the depth of the SHE needed for SPDZ is just *one*, making it very efficient.

Second Phase Costs

A careful analysis of the rest of the garbling phase of BMR-SPDZ implies that it requires six additional rounds of communication.[3]

Third Phase Costs

The online phase of the BMR-SPDZ protocol requires three rounds of interaction, one to open the secret shared values and two to verify the associated MACs.

Summary

In summary, the round complexity of BMR-SPDZ is $R_{\mathsf{ID}} + 10$ in the offline phase, and 3 in the online phase.

3.3 Extending the $\mathscr{F}_{\mathbf{FHE}}$/$\mathscr{F}_{\mathbf{SHE}}$ Functionalities

3.3.1 The Extended Functionality Definition

The first step in describing our new offline protocol for constructing the BMR circuit is to extend the functionalities $\mathscr{F}_{\mathrm{FHE}}$/$\mathscr{F}_{\mathrm{SHE}}$ to new functionalities $\mathscr{F}_{\mathrm{FHE}^+}$/ $\mathscr{F}_{\mathrm{SHE}^+}$. In Figure 3.2 we present the $\mathscr{F}_{\mathrm{FHE}^+}$ functionality, from which the definition of the $\mathscr{F}_{\mathrm{SHE}^+}$ functionality is immediate.

These new functionalities mimic the output possibilities of the SPDZ offline phase, which were exploited in [93] by allowing the functionality to produce encryptions of random data and encryptions of random bits. In addition the functionalities provide a version of **Output**, denoted by **Output+**, which does not allow the adversary to introduce an error value. There is also a new version of **InputData** called **InputData+** which will enable us to reduce the number of rounds of interaction in our main protocol. Functionally this does nothing different from **InputData** but it will be convenient to introduce a different name for a different implementation within our FHE functionality.

[3] With reference to [93] this is one round in the preprocessing-I phase and the start of the preprocessing-II phase due to the Output commands, and three to evaluate the required circuits in step 3 of preprocessing-II (since the circuits are of depth three, and hence require three rounds of computation), plus two to verify all the associated MAC values.

The Extended Functionality $\mathcal{F}_{\text{FHE}^+}$

This functionality runs the same **Initialize**, **Wait**, **InputData**, **Add**, **Multiply,** and **Output** commands as $\mathcal{F}_{\text{FHE}}$ of Figure 3.1. It additionally has the four following externally exposed commands:

Output+: On input (*output+,*varid*, i*) from all honest parties (if varid is present in memory), the functionality retrieves (varid,x), and if $i = 0$ it outputs (varid,x) to the adversary. The functionality then calls **Wait**, and only if **Wait** does not abort then outputs x to all parties if $i = 0$, or outputs x only to party i if $i \neq 0$.

InputData+: On input (*input+*,P_i,varid,x) from P_i and (*input+*,P_i,varid,?) from all other parties, with varid a fresh identifier, the functionality stores (varid,x). The functionality then calls **Wait**.

RandomElement: This command is executed on input (*randomelement*, varid) from all parties, with varid a fresh identifier. The functionality then selects uniformly at random $x \in \mathbb{F}_p$ and stores (varid,x).

RandomBit: This command is executed on input (*randombit*, varid) from all parties, with varid a fresh identifier. The functionality then selects uniformly at random $x \in \{0,1\}$ and stores (varid,x).

Figure 3.2: The Extended Functionality $\mathcal{F}_{\text{FHE}^+}$

3.3.2 Securely Realising the Extended Functionality

In Figure 3.3 we give the protocol Π_{FHE^+} for realising the $\mathcal{F}_{\text{FHE}^+}$ functionality in the $\mathcal{F}_{\text{FHE}}$-hybrid model. Let us start by looking at the **Output+** command in more detail (after first reading Figure 3.3). Suppose the adversary tries to make player P_j accept an incorrect value, by introducing errors into the calls to the weakly secure **Output** command from $\mathcal{F}_{\text{FHE}}$. The honest player P_j will receive $\mathsf{varid} + e_1$ instead of varid and $\mathsf{authvarid}_j + e_2$ instead of $\mathsf{authvarid}_j$, for some adversarially chosen values of e_1 and e_2. If player P_j is not to abort then these quantities must satisfy $\mathsf{authvarid}_j + e_2 = sk_j \cdot (\mathsf{varid} + e_1)$. Now since we know that $\mathsf{authvarid}_j = \mathsf{varid} \cdot sk_j$ then this implies that the adversary needs to select e_1 and e_2 such that $e_2 = sk_j \cdot e_1$, which it needs to do without having any knowledge of sk_j. Thus either the adversary needs to select $e_1 = e_2 = 0$, or he needs to guess the correct value of sk_j. This will happen with probability at most $1/p$, which is negligible.

We note that in the concurrent independent work of [48] a similar approach to our **Output+** command is taken in order to attain active security. Nevertheless, they use a global MAC key $\langle sk \rangle = \langle sk_1 \rangle + \cdots + \langle sk_n \rangle$ that is revealed to all parties after decryption, which means that sk needs to be renewed after each call to **Output+**. Thus, each call to their similar **Output+** implementation requires n calls to the expensive **InputData** protocol, which does not pay off in terms of concrete efficiency.

The protocol which implements **InputData+** works by first running **InputData** with a random value, and then later providing the difference between the random value input and the real input. This enables parallel *preprocessing* of the InputData procedure, thereby reducing the overall number of rounds.

The protocol which implements the **RandomElement** command generates an encrypted

Protocol Π_{FHE^+}

This protocol implements the functionality $\mathcal{F}_{\text{FHE}^+}$ in the $\mathcal{F}_{\text{FHE}}$-hybrid model.

Initialize: This performs the initialisation routine just as in the $\mathcal{F}_{\text{FHE}}$ functionality. However, in addition, each party executes **InputData** to obtain an encryption $\langle sk_i \rangle$ of a random MAC value sk_i known only to player P_i.

Output+: On input (*output+*, varid, i) from all honest parties, if varid is present in memory, the following steps are executed.

1. If $i \neq 0$, party P_i computes $\text{authvarid}_i = \langle \text{varid} \rangle \cdot \langle sk_i \rangle$, else, each party P_j computes $\text{authvarid}_j = \langle \text{varid} \rangle \cdot \langle sk_j \rangle$.
2. Parties call $\mathcal{F}_{\text{FHE}}$ with the command (*output*, varid, i).
3. If $i \neq 0$, parties call $\mathcal{F}_{\text{FHE}}$ with the command (*output,* authvarid_i*,* i), else, they use command (*output,* authvarid_j*,* j) for every $j \in [1,\ldots,n]$. Any party P_j aborts if $\text{authvarid}_j \neq \text{varid} \cdot sk_j$.

Depth Needed: $D(\text{varid}) + 1$. Round Cost: 2 (since the *output* calls in steps 2 and 3 can be performed in parallel).

InputData+: The first step of this command does not depend on the input, and so can be run in a preprocessing step if the number of values to be input per party are known in advance. Upon input $(input+, P_i, \text{varid}, x)$ with $x \in \mathbb{F}_p$ for P_i and $(input+, P_i, \text{varid}, ?)$ for all other parties:

1. Party P_i chooses a random $r_i \in \mathbb{F}_p$ (in the same field as x) and sends $(input, P_i, \text{varid-1}, r_i)$ to Functionality $\mathcal{F}_{\text{FHE}}$. All parties P_j with $j \neq i$ send $(input, P_i, \text{varid-1}, ?)$ to Functionality $\mathcal{F}_{\text{FHE}}$.
2. Party P_i broadcasts $c_i = x_i - r_i \pmod p$ to all parties.
3. Parties send $(add\text{-}scalar, c_i, \text{varid-1}, \text{varid})$ to Functionality $\mathcal{F}_{\text{FHE}}$.

Depth Needed: $D(x_i) = D(c) = 0$. Round Cost: $R_{\mathsf{ID}} + 1$. Although all R_{ID} rounds can be performed in parallel at the start of the protocol.

RandomElement:

1. For $i = 1,\ldots,n$, each P_i chooses a random $x_i \in \mathbb{F}_p$, and calls $\mathcal{F}_{\text{FHE}}$ with the command $(input, P_i, x_i)$ from party P_i and $(input, P_i, ?)$ for the others.
2. Call **Add** as many times as needed to compute $\langle x \rangle = \langle x_1 \rangle + \cdots + \langle x_n \rangle$.

Depth Needed: $D(x_i) = \max\{D(x_i)\} = 0$. Round Cost: R_{ID}.

RandomBit: This command requires a more elaborate implementation:

1. For $i = 1,\ldots,n$, call $\mathcal{F}_{\text{FHE}}$ with the command $(input, P_i, x_i)$ from party P_i and $(input, P_i, ?)$ for the rest of the parties.
2. Call **Add** as many times as needed to compute $\langle x \rangle = \langle x_1 \rangle + \cdots + \langle x_n \rangle$.
3. Call **Multiply** to compute $\langle s \rangle = \langle x \rangle \cdot \langle x \rangle$.
4. Call $\mathcal{F}_{\text{FHE}^+}$ on input (*output+*, s, 0) so all parties obtain s. If $s = 0$ then restart the protocol.
5. Parties compute $y = \sqrt{s} \pmod p$. Between the two possible outputs, y is chosen according to a previously agreed method (e.g. the biggest one when looked as an integer).
6. Call **Add-scalar** and **Multiply-scalar** to compute $\langle \text{varid} \rangle = (1 + \langle x \rangle / y)/2$.

Depth Needed: $D(s) + 1 = 2$. Note the output encrypted bit has depth zero. Round Cost: $R_{\mathsf{ID}} + 2$.

Figure 3.3: Protocol Π_{FHE^+}.

random value $\langle x \rangle$, unknown to any party as long as one of the parties honestly chooses his additional share x_i randomly.

The protocol which implements the **RandomBit** command is more elaborate, and borrows much from the equivalent operations in the SPDZ offline phase, see [44]. The basic idea is to generate an encrypted random value $\langle x \rangle$, unknown to any party. This value is then squared to obtain $\langle s \rangle$. The value of s is then publicly revealed and an arbitrary square root y is taken, according to any previously agreed rule. As long as $s \neq 0$ (which happens with negligible probability due to the size of p) we then have that $\langle b \rangle = \langle x \rangle / y$ is an encryption of a value chosen uniformly from $\{-1, 1\}$. Since p is prime, with probability 1/2 the square root taken will be equal to x and with probability 1/2 it will be equal to $-x$. This encryption of a value in $\{-1, 1\}$ is turned into an encryption of a value in $\{0, 1\}$ by the final step, by computing $(\langle b \rangle + 1)/2$, which is a linear function and can be thus computed by calling **Add-Scalar** and **Multiply-Scalar**. However, unlike in SPDZ no sacrificing procedure is required as the **Output+** command is actively secure.

Theorem 3.3.1. *Protocol Π_{FHE^+} securely implements $\mathscr{F}_{\text{FHE}^+}$ in the $\mathscr{F}_{FHE}$-hybrid model in the UC framework, in the presence of static, active adversaries corrupting up to $t \leq n-1$ parties.*

Proof (sketch) By [36], it suffices to prove the security of Protocol Π_{FHE^+} in the SUC (simple UC) framework. We will sketch the proof for each of the processes in the functionality separately. In the $\mathscr{F}_{\text{FHE}}$-hybrid model the security follows in a straightforward way utilizing the security of the commands in $\mathscr{F}_{\text{FHE}}$.

Output+: The security of **Output+** relies on the security of the **InputData**, **Multiply** and **Output** commands of $\mathscr{F}_{\text{FHE}}$. Namely, by the security of **InputData** we have that all sk_j values are secret, and by the security of **Output** the only change that $\mathscr{A}$ can make to the output is an additive difference e (fixed before the output is given). Thus, $\mathscr{A}$ can only change the output if it chooses additive differences e_1, e_2 with $e_1 \neq 0$ such that $(x + e_1) \cdot sk_j = x \cdot sk_j + e_2 \pmod{p}$, where x is the value output. This implies that $e_1 \cdot sk_j = e_2 \pmod{p}$. Since sk_j is secret, the adversary can cause this equality to hold with probability at most p.

We remark that the MAC key sk_j is only used for output values given to P_j. Thus, it always remains secret (even when used for many outputs).

The simulator for **Output+** works simply by simulating the **Output** interaction with $\mathscr{F}_{\text{FHE}}$ for all honest P_i. Regarding a corrupt P_j, the simulator receives the value x that is supposed to be output. Furthermore, the simulator receives the value sk_j from the **InputData** instruction, as well as any errors that are introduced in the **Output** calls by corrupted parties. Thus, the simulator can construct the exact value that $\mathscr{A}$ would receive in a real execution.

InputData+: The only difference between **InputData+** and **InputData** is that **InputData+** can be run such that the actual input is only known to the party in the last round of

the protocol. This is done in a straightforward way by using **InputData** to have a party input a random string, and then using that result to mask the real data (at the end). The simulator for this procedure therefore relies directly on the **InputData** procedure of $\mathscr{F}_{\text{FHE}}$ in a straightforward way. Namely, in the $\mathscr{F}_{\text{FHE}}$-hybrid model when the party P_i is corrupted, the simulator receives the value r_i that party P_i sends to **InputData**. Then, upon receiving c_i as broadcast by P_i, the simulator defines $x_i = c_i + r_i \pmod p$ and sends $(input+, P_i, \mathsf{varid}, x_i)$ to the ideal functionality as input. In the case that P_i is honest, the simulator chooses a random $c_i \in \mathbb{F}_p$ and simulates P_i broadcasting that value. Furthermore, it simulates the $(input, ...)$ and $(add\text{-}scalar, ...)$ interaction with $\mathscr{F}_{\text{FHE}}$.

The view of the adversary is identical in the simulated and real executions. In addition, since **InputData** is secure and c_i is broadcast and therefore the same for all parties, the protocol fully determines the input value $x_i = c_i + r_i \pmod p$, as required.

RandomElement: This is a straightforward coin tossing protocol. The security is derived from the fact that $\mathscr{F}_{\text{FHE}}$ provides a secure **InputData** protocol that reveals no information about the input values. Thus, no party knows anything about the x-values input by the others. Formally, a simulator just simulates the message interaction with $\mathscr{F}_{\text{FHE}}$ for all of the $(input, P_i, x_i)$ and $(input, P_i, ?)$ messages. As long as at least one party is honest, the distribution over the value x defined is uniform, as required.

RandomBit: The first step of this protocol is to essentially run **RandomElement** in order to define a random shared value x. Then, the value $s = x^2 \pmod p$ is output to all parties, and each takes the same square-root y of s. Assume that the square root taken is the one that is between 1 and $(p-1)/2$. Now, if $1 \le x \le \frac{p-1}{2}$, then $y = x$ and so $\langle b \rangle = \langle 1 \rangle$, and we have that $\langle \mathsf{varid} \rangle = \langle \frac{1+1}{2} \rangle = \langle 1 \rangle$. Else, if $\frac{p-1}{2} < x \le p-1$ then $\langle b \rangle = \langle -1 \rangle$ and we have $\langle \mathsf{varid} \rangle = \langle \frac{-1+1}{2} \rangle = \langle 0 \rangle$. The security relies on the fact that the result is *fully determined* from the $(input, ...)$ messages sent in the beginning. Relying on the security of **InputData** and **Add/Multiply** in $\mathscr{F}_{\text{FHE}}$, and on the security of the **Output+** procedure, the value x is uniformly distributed and the value s that is output to all parties equals x^2 and no other value. All other steps are deterministic and thus this guarantees that the output is a uniformly distributed bit, as required.

Regarding simulation, the simulator simulates the calls to **InputData**, **Add** and **Multiply** as in the protocol. For the output, the simulator simply chooses a random s as the value received from **Output+**. The view of the parties is clearly identical to in a real execution.

■

3.4 The First Variant of the BMR-SHE Protocol, $\Pi_{\text{Depth}-4}$

In this section we outline our basic protocol, which follows much upon the lines of the BMR-SPDZ protocol. The modifications needed for a variant using only depth three will be left to Section 3.5. We divide our discussion into four subsections. In the first one we outline the offline functionality $\mathcal{F}_{\text{Preprocessing}}$ required for our specific garbling scheme. The next subsection discuss how to implement the overall BMR-style protocol assuming said functionality. In the two last subsections we discuss a protocol to implement $\mathcal{F}_{\text{Preprocessing}}$ and analyse its efficiency.

3.4.1 Functionality $\mathcal{F}_{\text{Preprocessing}}$ for the Offline Phase

We first present the offline functionality (see Figure 3.4) for our main MPC protocol. This is almost identical to the offline functionality for the BMR-SPDZ protocol of [93]. The main difference is that it is built on top of our $\mathcal{F}_{\text{FHE}^+}$ functionality from the previous section, as opposed to the SPDZ MPC protocol. In particular this means we have just a single preprocessing step as opposed to the two phases in [93], which are in turn inherited from the two phases of the SPDZ protocol.

We recall from Section 2.6 that the functionality also has an additional task: Revealing the output wire masks towards all parties and the input wire masks towards the relevant input party. Everything else in the garbled circuit remains encrypted as in $\mathcal{F}_{\text{FHE}^+}$.

3.4.2 The BMR-SHE Protocol Specification $\Pi_{\text{MPC},4}$

We can now give our protocol $\Pi_{\text{MPC},4}$, described in Figure 3.5, which securely computes the functionality $\mathcal{F}_{\text{MPC}}$ described in Figure 2.1 in the $\mathcal{F}_{\text{Preprocessing}}$-hybrid model. The computational and communication costs of $\Pi_{\text{MPC},4}$ are mainly in the preprocessing step, with the online phase adding only a depth of one to the SHE scheme and two more rounds of communication, all of these costs coming from the need for an actively secure **Output+**. As the online phase follows the style of the one in SPDZ-BMR [93] or, more broadly the one of BMR which was described in Section 2.6.1, we do not discuss it in details here.

3.4.3 The $\Pi_{\text{Preprocessing},4}$ Protocol

For completeness, we show how to calculate the output indicators for functions $f_g = AND$ and $f_g = XOR$ in Figure 3.7 as shown in [93]. Note that we consume a multiplicative depth of two for both operations.

- For $f_g = AND$, we compute $\langle t \rangle = \langle \lambda_a \rangle \cdot \langle \lambda_b \rangle$ and then $\langle x_A \rangle = (\langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_B \rangle = (\langle \lambda_a \rangle - \langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_C \rangle = (\langle \lambda_b \rangle - \langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_D \rangle = (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle + \langle t \rangle - \langle \lambda_c \rangle)^2$.
- For $f_g = XOR$, we first compute $\langle t \rangle = \langle \lambda_a \rangle \oplus \langle \lambda_b \rangle = \langle \lambda_a \rangle + \langle \lambda_b \rangle - 2 \cdot \langle \lambda_a \rangle \cdot \langle \lambda_2 \rangle$, and then $\langle x_A \rangle = (\langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_B \rangle = (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle + 2 \cdot \langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_C \rangle = \langle x_B \rangle$, $\langle x_D \rangle = \langle x_A \rangle$.

Functionality $\mathcal{F}_{\text{Preprocessing}}$

This functionality runs the same commands as $\mathcal{F}_{\text{FHE}^+}$. In addition it has the following command:

Preprocessing: On input $(\texttt{Garbling}, C_f)$ from all parties, where C_f is a boolean circuit, denote by W its set of wires and by G its set of AND gates. The functionality performs the following operations:

- For all wires $w \in [1,\ldots,W]$:
  - The functionality stores a random mask λ_w, where $\lambda_w \in \{0,1\}$.
  - For every value $\beta \in \{0,1\}$, each party P_i chooses and stores a random key $k^i_{w,\beta} \in \mathbb{F}_p$.
- For all input wires w where a party P_i is meant to provide input, the functionality reveals λ_w to that party by running **Output+** as in $\mathcal{F}_{\text{FHE}^+}$.
- For all output wires w the functionality reveals λ_w to all parties by running **Output+** as in $\mathcal{F}_{\text{FHE}^+}$.
- For every gate g with input wires $1 \le a,b \le W$ and output wire $1 \le c \le W$.
  - Party P_i provides the following inputs for $x \in \{a,b\}, g \in [1,\ldots,G]$ by running **Input-Data** as in $\mathcal{F}_{\text{FHE}^+}$:

$$F_{k^i_{x,0}}(0||1||g),\ldots,F_{k^i_{x,0}}(0||n||g), \qquad F_{k^i_{x,0}}(1||1||g),\ldots,F_{k^i_{x,0}}(1||n||g)$$
$$F_{k^i_{x,1}}(0||1||g),\ldots,F_{k^i_{x,1}}(0||n||g), \qquad F_{k^i_{x,1}}(1||1||g),\ldots,F_{k^i_{x,1}}(1||n||g)$$

    (In our protocols, the parties actually provide sums of pairs of these values, see Figure 3.7. This reduces the number of values input from 8 per-party per-gate to only 4 per-party per-gate.)
  - Define the selector variables

$$\chi_1 = (f_g(\lambda_a,\lambda_b)-\lambda_c)^2.$$
$$\chi_2 = (f_g(\lambda_a,\bar{\lambda_b})-\lambda_c)^2.$$
$$\chi_3 = (f_g(\bar{\lambda_a},\lambda_b)-\lambda_c)^2.$$
$$\chi_4 = (f_g(\bar{\lambda_a},\bar{\lambda_b})-\lambda_c)^2.$$

  - Set $A_g = (A^1_g,\ldots,A^n_g)$, $B_g = (B^1_g,\ldots,B^n_g)$, $C_g = (C^1_g,\ldots,C^n_g)$, $D_g = (D^1_g,\ldots,D^n_g)$ where for $1 \le j \le n$:

$$A^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(0||j||g) + F_{k^i_{b,0}}(0||j||g)\right) + k^j_{c,\chi_1}$$
$$B^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(1||j||g) + F_{k^i_{b,1}}(0||j||g)\right) + k^j_{c,\chi_2}$$
$$C^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,1}}(0||j||g) + F_{k^i_{b,0}}(1||j||g)\right) + k^j_{c,\chi_3}$$
$$D^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,1}}(1||j||g) + F_{k^i_{b,1}}(1||j||g)\right) + k^j_{c,\chi_4}$$

  - The functionality finally stores the values A_g, B_g, C_g, D_g for all g.

Figure 3.4: The Preprocessing Functionality $\mathcal{F}_{\text{Preprocessing}}$.

The MPC Protocol - $\Pi_{\text{MPC},4}$

On input a boolean circuit C_f representing the function f, parties execute the following commands in sequence:

Preprocessing: This sub-task is performed as follows.

- Call **Initialize** on $\mathscr{F}_{\text{Preprocessing}}$ to initialize the FHE scheme.
- Call **Preprocessing** on $\mathscr{F}_{\text{Preprocessing}}$ with input C_f.

Online Computation: This sub-task is performed as follows.

- For all his input wires w, each party computes $\Lambda_w = \rho_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage, and ρ_w is his input on that wire. Λ_w is broadcast to all parties.
- Party P_i calls **Output+** on $\mathscr{F}_{\text{Preprocessing}}$ with all parties as receivers, in order to reveal them his keys $\langle k^i_{w,\Lambda_w} \rangle$ associated to Λ_w, for all his input wires w.
- The parties call **Output+** on $\mathscr{F}_{\text{Preprocessing}}$ to decrypt A_g, B_g, C_g and D_g for every gate g.
- Passing through the circuit topologically, the parties can now locally compute the following operations for each gate g. Let the gates input wires be labelled a and b, and the output wire be labelled c.
  - For $j = 1, \ldots, n$ compute k^j_c according to the following cases:
    $(\Lambda_a, \Lambda_b) = (0,0)$: Set $k^j_c = A^j_g - \left(\sum_{i=1}^n F_{k^i_a}(0||j||g) + F_{k^i_b}(0||j||g)\right)$.
    $(\Lambda_a, \Lambda_b) = (0,1)$: Set $k^j_c = B^j_g - \left(\sum_{i=1}^n F_{k^i_a}(1||j||g) + F_{k^i_b}(0||j||g)\right)$.
    $(\Lambda_a, \Lambda_b) = (1,0)$: Set $k^j_c = C^j_g - \left(\sum_{i=1}^n F_{k^i_a}(0||j||g) + F_{k^i_b}(1||j||g)\right)$.
    $(\Lambda_a, \Lambda_b) = (1,1)$: Set $k^j_c = D^j_g - \left(\sum_{i=1}^n F_{k^i_a}(1||j||g) + F_{k^i_b}(1||j||g)\right)$.
  - If $k^i_c \notin \{k^i_{c,0}, k^i_{c,1}\}$, then P_i outputs `abort`. Otherwise, it proceeds. If P_i aborts it notifies all other parties with that information. If P_i is notified that another party has aborted it aborts as well.
  - If $k^i_c = k^i_{c,0}$ then P_i sets $\Lambda_c = 0$ otherwise if $k^i_c = k^i_{c,1}$ then P_i sets $\Lambda_c = 1$.
  - The output of the gate is defined to be $(k^1_c, \ldots, k^n_c)$ and Λ_c.
- Assuming party P_i does not abort it will obtain Λ_w for every circuit-output wire w. The party can then recover the actual output value from $\rho_w = \Lambda_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

Depth Needed: $D(Output + (\{A_g\}, \{B_g\}, \{C_g\}, \{D_g\})) = 3 + 1 = 4$.

Round Cost: The round cost of the online stage is that of the first three steps, which can be done in parallel in two rounds.

Figure 3.5: The MPC Protocol - $\Pi_{\text{MPC},4}$.

The Offline Protocol – $\Pi_{\text{Preprocessing},4}$

The protocol runs the commands **Initialize**, **InputData**, **InputData+**, **Wait**, and **Output+** by calling the equivalent commands on $\mathcal{F}_{\text{FHE}^+}$. Thus we only need to implement the additional **Preprocessing** command of $\mathcal{F}_{\text{Preprocessing}}$ as follows:

1. Call **Initialize** on the functionality $\mathcal{F}_{\text{FHE}^+}$ with input a prime $p > 2^k$.
2. **Generate wire masks:** For every circuit wire w we need to generate a random and hidden masking-values λ_w. Thus for *all* wires w the parties execute **RandomBit** of $\mathcal{F}_{\text{FHE}^+}$, the output is denoted by $\langle \lambda_w \rangle$.
   Depth Needed: $D(RandomBit) = 2$
   Round Cost: $R_{\mathsf{ID}} + 2$.
3. **Generate keys:** For every wire w, each party $i \in [1, \ldots, n]$ and for $\beta \in \{0,1\}$, the parties execute the command **InputData** of the functionality $\mathcal{F}_{\text{FHE}^+}$ to obtain output $\langle k^i_{w,\beta} \rangle$, where P_i learns $k^i_{w,\beta}$. We shall denote vector of keys $\left(\langle k^i_{w,\beta} \rangle\right)_{i=1}^n$ by $\langle \mathbf{k}_{w,\beta} \rangle$.
   Depth Needed: $D(k^i_{w,\beta}) = 0$.
   Round Cost: R_{ID}.
4. **Reveal masks for circuit-input-wires:** For all input wires w where a party P_i is meant to provide input, execute the command **Output+** in $\mathcal{F}_{\text{FHE}^+}$ to decrypt $\langle \lambda_w \rangle$ to that party.
   Depth Needed: $\max(D(RandomBit), D(Output + (\lambda_w))) = \max(2,1) = 2$.
   Round Cost: 2.
5. **Reveal masks for circuit-output-wires:** In order to reveal the real values of the circuit-output-wires it is required to reveal their masking values. That is, for every circuit-output-wire w, the parties execute the command **Output+** on the functionality $\mathcal{F}_{\text{FHE}^+}$ for the stored value $\langle \lambda_w \rangle$.
   Depth Needed: $\max(D(RandomBit), D(Output + (\lambda_w))) = \max(2,1) = 2$.
   Round Cost: 2.
6. **Calculate garbled gates:** See Figure 3.7 for the details of this step.

We note that steps two and three can be run in parallel, and that steps four and five also can be run in parallel, but need to follow step two. We also note that the calls to **InputData+** in the last step (detailed in Figure 3.7) need to be executed after step three. Hence, we have:
Total Depth Needed: 3.
Total Round Cost: $\max(R_{\mathsf{ID}} + 3, R_{\mathsf{ID}} + 4) = R_{\mathsf{ID}} + 4$.

Figure 3.6: The Preprocessing Protocol: $\Pi_{\text{Preprocessing},4}$.

Calculate Garbled Gates Step of $\Pi_{\text{Preprocessing},4}$

This step is operated for each gate g in the circuit in parallel. Specifically, let g be a gate whose input wires are a, b and output wire is c. Do as follows:

(a) **Calculate external values:** This step calculates the external values $\langle x_A\rangle, \langle x_B\rangle, \langle x_C\rangle, \langle x_D\rangle$ corresponding to each row of the garbled gate. Each of them is either $\langle 0\rangle$ or $\langle 1\rangle$ and is determined according to some quadratic function f_g on $\langle\lambda_a\rangle, \langle\lambda_b\rangle, \langle\lambda_c\rangle$, depending on the truth table of the gate. See Section 3.4.3 for details.

$$\langle x_A\rangle = (f_g(\langle\lambda_a\rangle, \langle\lambda_b\rangle) - \langle\lambda_c\rangle)^2 \qquad \langle x_B\rangle = (f_g(\langle\lambda_a\rangle, (1 - \langle\lambda_b\rangle)) - \langle\lambda_c\rangle)^2$$
$$\langle x_C\rangle = (f_g((1 - \langle\lambda_a\rangle), \langle\lambda_b\rangle) - \langle\lambda_c\rangle)^2 \qquad \langle x_D\rangle = (f_g((1 - \langle\lambda_a\rangle), (1 - \langle\lambda_b\rangle)) - \langle\lambda_c\rangle)^2$$

Depth Needed: $D(x_*) = D(\lambda_*) + 2 = 2$.

(b) **Assign the correct vector:** The external values are used to choose, for every row of garbled gate, either $\mathbf{k}_{c,0}$ or $\mathbf{k}_{c,1}$, for $t = A, B, C, D$,

$$\langle \mathbf{v}_{c,x_t}\rangle = (1 - \langle x_t\rangle) \cdot \langle \mathbf{k}_{c,0}\rangle + \langle x_t\rangle \cdot \langle \mathbf{k}_{c,1}\rangle.$$

Depth Needed: $D(\mathbf{v}_{c,x_*}) = \max(D(x_*), D(\mathbf{k}_{c,*})) + 1 = 3$.

(c) **Calculate garbled gate rows:** Party i can now compute the $2 \cdot n$ PRF values $F_{k^i_{w,\beta}}(0||1||g), \ldots, F_{k^i_{w,\beta}}(0||n||g)$ and $F_{k^i_{w,\beta}}(1||1||g), \ldots, F_{k^i_{w,\beta}}(1||n||g)$, for each input wire w of gate G, and $\beta = 0, 1$.

$$F^0_{k^i_{w,\beta}}(g) = \left(F_{k^i_{w,\beta}}(0||1||g), \ldots, F_{k^i_{w,\beta}}(0||n||g)\right)$$
$$F^1_{k^i_{w,\beta}}(g) = \left(F_{k^i_{w,\beta}}(1||1||g), \ldots, F_{k^i_{w,\beta}}(1||n||g)\right).$$

Then, they call $4 \cdot n \cdot G$ times the command **InputData+** on the functionality $\mathcal{F}_{\text{FHE}}$, so all the parties obtain the output:

$$\langle F^0_{k^i_{a,0}} + F^0_{k^i_{b,0}}\rangle, \quad \langle F^1_{k^i_{a,0}} + F^0_{k^i_{b,1}}\rangle, \quad \langle F^0_{k^i_{a,1}} + F^1_{k^i_{b,0}}\rangle, \quad \langle F^1_{k^i_{a,1}} + F^1_{k^i_{b,1}}\rangle.$$

All the parties now compute the four rows $\langle A_g\rangle, \langle B_g\rangle, \langle C_g\rangle, \langle D_g\rangle$ for every garbled gate g via

$$\langle A_g\rangle = \langle \mathbf{v}_{c,x_A}\rangle + \sum_{i=1}^{n} \langle F^0_{k^i_{a,0}}(g) + F^0_{k^i_{b,0}}(g)\rangle \qquad \langle B_g\rangle = \langle \mathbf{v}_{c,x_B}\rangle + \sum_{i=1}^{n} \langle F^1_{k^i_{a,0}}(g) + F^0_{k^i_{b,1}}(g)\rangle$$
$$\langle C_g\rangle = \langle \mathbf{v}_{c,x_C}\rangle + \sum_{i=1}^{n} \langle F^0_{k^i_{a,1}}(g) + F^1_{k^i_{b,0}}(g)\rangle \qquad \langle D_g\rangle = \langle \mathbf{v}_{c,x_D}\rangle + \sum_{i=1}^{n} \langle F^1_{k^i_{a,1}}(g) + F^1_{k^i_{b,1}}(g)\rangle$$

Round Cost: $R_{ID} = R_{ID} + 1$, but the R_{ID} can be done in parallel before. Depth Needed: $D(A_g) = D(B_g) = D(C_g) = D(D_g) = D(\mathbf{v}_{c,x_*}) = 3$.

Figure 3.7: Calculate Garbled Gates Step of $\Pi_{\text{Preprocessing},4}$.

Claim 3.4.1. *Protocol* $\Pi_{\text{Preprocessing},4}$ *securely implements* $\mathcal{F}_{\text{Preprocessing}}$ *in the* $\mathcal{F}_{\text{FHE}^+}$*-hybrid model in the UC framework, in the presence of static, active adversaries corrupting* $t \leq n-1$ *parties.*

Proof (sketch) We have modelled $\mathcal{F}_{\text{Preprocessing}}$ in almost the same way as in [93]. The security of our protocol follows from the proof of the security of the BMR-SPDZ protocol there: We use our extension of Gentry's MPC protocol as opposed to the SPDZ protocol, which is purely an implementation change. ■

3.4.4 Analysis of Efficiency

Just as in our analysis of the BMR-SPDZ protocol in Section 3.2.3, we wish to estimate the cost of the most expensive operations, which are the encryptions of (random or not) input data.

- Each party calls **InputData** once during the **Initialize** phase of the extended FHE functionality.
- We perform W **RandomBit** operations, each of which consumes a C_{ID} per party.
- To create the encrypted PRF keys we require an additional $2 \cdot W$ invocations of C_{ID} per party.
- Finally, in order to compute the rows of the garbled gate we require $4 \cdot n^2 \cdot G$ invocations of **InputData**, which correspond to $4 \cdot n \cdot G$ invocations of per party.

Thus, we conclude that the cost of encrypting the data for the depth-4 BMR-SHE protocol is given by the expression $\left(4 \cdot n^2 \cdot G + (3 \cdot W + 1) \cdot n\right) \cdot C_{\text{ID}}$, which is *quadratic* in n as opposed to the *cubic* complexity in the number of parties of the BMR-SPDZ protocol.

3.5 A Lower Depth Variant of the BMR-SHE Protocol, $\Pi_{\text{Depth}-3}$

In this section we give a description of the protocol $\Pi_{\text{Depth}-3}$ (see Figure 3.8 and Figure 3.9 for the offline and online modifications respectively) which requires only a multiplicative depth of three rather than four as in $\Pi_{\text{Depth}-4}$. This reduction on the depth of the SHE scheme comes directly from the reduction of the depth of the circuit used for garbling the actual circuit to evaluate. On the downside, we require additional $2 \cdot W \cdot n \cdot (n-1)$ calls to **InputData** and some more multiplications in the offline and online phases. The new protocol $\Pi_{\text{Depth}-3}$ is, in fact, just a variant of $\Pi_{\text{Depth}-4}$. It remains to be empirically tested for which set of parameters one would be preferred in practice one over the other.

3.5.1 Protocol $\Pi_{\text{Depth}-3}$ Description

Our earlier protocol $\Pi_{\text{Depth}-4}$ securely computes the BMR garbled gates as follows: For every gate the parties first compute the shares $\langle x_A \rangle, \langle x_B \rangle, \langle x_C \rangle, \langle x_D \rangle$ and then use these shares to compute the shares $\langle \mathbf{v}_{c,x_A} \rangle, \langle \mathbf{v}_{c,x_B} \rangle, \langle \mathbf{v}_{c,x_C} \rangle, \langle \mathbf{v}_{c,x_D} \rangle$ of the keys $\mathbf{k}_{c,0}$ or $\mathbf{k}_{c,1}$ on the output wire of the gate. Finally, these are masked by the pseudorandom values provided by all parties, see Figure 3.7. Observing how these equations are computed, we can verify that obtaining the $\langle x_* \rangle$ values require two multiplications and computing $\langle \mathbf{v}_{c,x_*} \rangle$ requires an additional multiplication. The final multiplication, making the overall protocol depth-4, is needed for computing **Output+**. The goal of the variant here presented is to produce the $\langle \mathbf{v}_{c,x_*} \rangle$ values directly, with just two multiplications instead of three.

In order to achieve this, we limit our considerations to AND and XOR gates, for which we provide formulae computing directly $\langle \mathbf{v}_{c,x_*} \rangle$. The main idea is that it actually suffices to store in that vector either the key $\mathbf{k}_{c,*}$ or its opposite $-\mathbf{k}_{c,*}$ modulo p. The reason that this suffices is that the *square* of these values is the same. Thus, we have two versions of each key, which we denote the *basic-key* and the *squared-key*. This introduces two main modifications in the offline protocol:

1. Parties call **RandomElement** in order to generate each basic-key. They jointly and obliviously square the result, obtaining a squared-key which is revealed to the appropriate party.

2. Afterwards, parties can compute $\langle \mathbf{v}_{c,x_*} \rangle$, which equals the basic-key (or its opposite) on wire c. The result is masked with the outputs of the PRFs, which are keyed under the revealed squared-keys.

Observe that then, in the online phase, the basic-key or its opposite is revealed when evaluating the garbled circuit. The parties then square it in order to get the square-key and compute the PRF values enabling the decryption of the next garbled gate. Since the basic-key is random and was *never revealed*, the parties have no idea of whether they received the basic-key or it's opposite. Otherwise this would leak information about the values on the wires, as the 'sign choice' in $\langle \mathbf{v}_{c,x_*} \rangle$ depends on the wire masks, which we show in the next paragraphs.

Note that these modifications add $2 \cdot W \cdot n \cdot (n-1)$ calls to **InputData**, in order to generate the basic-keys via calls to **RandomElement**.

3.5.1.1 The AND Gate

We now present the equations for computing an AND gate with input wires a, b and output wire c. We denote the basic-keys (i.e not squared) on the output wire c by $\tilde{\mathbf{k}}_{c,0}, \tilde{\mathbf{k}}_{c,1}$.

In order to motivate these equations, we explain in details the first of them, Equation 3.1, which computes $\langle \mathbf{v}_{c,x_A} \rangle$ and corresponds to the first ciphertext in the garbled gate. We know

The Lower Depth Offline Protocol – $\Pi_{\text{Preprocessing,3}}$

This protocol is identical to the $\Pi_{\text{Preprocessing,4}}$ protocol given in Figure 3.6, except for the following changes:

3 **Generate keys** in Figure 3.6 is changed as follows:

a) For every wire w, bit value $\beta \in \{0,1\}$ and party $i \in [1,\ldots,n]$, the parties execute the command **RandomElement** of the functionality $\mathscr{F}_{\text{FHE}^+}$ to obtain output $\langle \tilde{k}^i_{w,\beta} \rangle$. We stress that nobody learns $\tilde{k}^i_{w,\beta}$. Let varid be the identifier of $\langle \tilde{k}^i_{w,\beta} \rangle$. In the following, we shall abuse the notation to denote $\langle \tilde{\mathbf{k}}_{w,\beta} \rangle = \left(\langle \tilde{k}^1_{w,\beta} \rangle, \ldots, \langle \tilde{k}^n_{w,\beta} \rangle \right)$.

b) The parties call $(multiply, \text{varid}, \text{varid}, \text{varid2})$ where varid2 is a new identifier, in order to share a ciphertext $\langle k^i_{w,\beta} \rangle = \langle \tilde{k}^i_{w,\beta} \rangle^2$.

c) The parties call $\mathscr{F}_{\text{FHE}^+}$ on input $(output+, \text{varid2}, i)$ for party P_i to obtain $k^i_{w,\beta}$.

Depth Needed: $D(Output + (k^i_{w,\beta})) = 2$.
Round Cost: $R_{\text{ID}} + 2$.

4 **Calculate garbled gates** in Figure 3.7 is changed as follows:

a) The **calculate external values** and **assign the correct vector** phases are replaced by the following functions, that choose, for every row in the garbled gate, either $\tilde{\mathbf{k}}_{c,0}$, $-\tilde{\mathbf{k}}_{c,0}$, $\tilde{\mathbf{k}}_{c,1}$ or $-\tilde{\mathbf{k}}_{c,1}$.

- For an AND gate, the parties compute shares of the keys on the output wires according to Equations (3.1)–(3.4).
- For a XOR gate, the parties compute shares of the keys on the output wires according to Equations (3.5)–(3.7).

Depth Needed: $D(\mathbf{v}_{c,x_A}) = D(\mathbf{v}_{c,x_B}) = D(\mathbf{v}_{c,x_C}) = D(\mathbf{v}_{c,x_D}) = 2$.

Total Round Cost: $\max(R_{\text{ID}} + 3, R_{\text{ID}} + 4) = R_{\text{ID}} + 4$.
Total Depth Needed: 2.

Figure 3.8: The Modified Preprocessing Protocol $\Pi_{\text{Preprocessing,3}}$.

The modified MPC Protocol - $\Pi_{\text{MPC,3}}$

This protocol is identical to the $\Pi_{\text{MPC,4}}$ protocol described in Figure 3.5, except for the four cases of the **Online Computation** sub-task, in which for $j = 1,\ldots,n$, the values k^j_c are now computed as follows:

Case $(\Lambda_a, \Lambda_b) = (0,0)$: Compute $k^j_c = \left(A^j_g - (\sum_{i=1}^n F_{k^i_a}(0||j||g) + F_{k^i_b}(0||j||g)) \right)^2$.

Case $(\Lambda_a, \Lambda_b) = (0,1)$: Compute $k^j_c = \left(B^j_g - (\sum_{i=1}^n F_{k^i_a}(1||j||g) + F_{k^i_b}(0||j||g)) \right)^2$.

Case $(\Lambda_a, \Lambda_b) = (1,0)$: Compute $k^j_c = \left(C^j_g - (\sum_{i=1}^n F_{k^i_a}(0||j||g) + F_{k^i_b}(1||j||g)) \right)^2$.

Case $(\Lambda_a, \Lambda_b) = (1,1)$: Compute $k^j_c = \left(D^j_g - (\sum_{i=1}^n F_{k^i_a}(1||j||g) + F_{k^i_b}(1||j||g)) \right)^2$.

Depth Needed: $D_{Out+} + D(\{A_g\},\{B_g\},\{C_g\},\{D_g\}) = 2 + 1 = 3$.

Figure 3.9: The Modified Protocol $\Pi_{\text{MPC,3}}$.

from Sections 2.5 and 2.6 that those correspond to external values $(\Lambda_a, \Lambda_b) = (0,0)$. We break our explanation in two cases below.

First, if the actual value on wire a equals 0, then $\langle \lambda_a \rangle = \Lambda_a + 0 = 0$, and so $\langle \mathbf{v}_{c,x_A} \rangle$ is a function of the first row of the equation. Knowing that the actual value on a is 0, then the actual output equals 0 irrespective of b, since this is an AND gate. Thus, if the wire mask on c equals 0 then the output has to be $\langle \tilde{\mathbf{k}}_{c,0} \rangle$ and otherwise it should be $\langle \tilde{\mathbf{k}}_{c,1} \rangle$, as the equation ensures.

In the opposite case, when the actual value on wire a equals 1, then the value of $\langle \mathbf{v}_{c,x_A} \rangle$ depends only on the second row of the equation (since $1 - \langle \lambda_a \rangle = 0$). More precisely, it depends on the actual value on wire b (which equals λ_b, as $\Lambda_b = 0$) and the wire mask λ_c on the output wire. If $b = 1$ and $\lambda_c = 0$ or if $b = 0$ and $\lambda_c = 1$ then it should be that $\langle \mathbf{v}_{c,x_A} \rangle = \pm \langle \tilde{\mathbf{k}}_{c,1} \rangle$ (since in the first case $a = b = 1$ and the actual output is 1, whereas in the second case the actual output is 0 but $\lambda_c = 1$, so $\Lambda_c = 1$). We achieve this goal by multiplying $\langle \tilde{\mathbf{k}}_{c,1} \rangle$ by $\langle \lambda_b \rangle - \langle \lambda_c \rangle$ which equals ± 1 in both of these cases (and 0 otherwise). We then multiply $\langle \tilde{\mathbf{k}}_{c,0} \rangle$ by $1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle$, which equals 0 in both of these cases $b = 0, \lambda_c = 1$ and $b = 1, \lambda_c = 0$. In contrast, if $b = \lambda_c = 0$ or $b = \lambda_c = 1$ then the output should be $\pm \langle \tilde{\mathbf{k}}_{c,0} \rangle$ (as above, because if $b = \lambda_c = 0$ then the actual output is 0, and if $b = \lambda_c = 1$ then the actual output is 1 but the external value is 0). Finally we obtain next equation:

$$\begin{aligned} \langle \mathbf{v}_{c,x_A} \rangle &= (1 - \langle \lambda_a \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \\ &\quad + \langle \lambda_a \rangle \cdot \left((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right). \end{aligned} \tag{3.1}$$

The remaining three equations are computed similarly, as follows:

$$\begin{aligned} \langle \mathbf{v}_{c,x_B} \rangle &= (1 - \langle \lambda_a \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \\ &\quad + \langle \lambda_a \rangle \cdot \left((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \right) \end{aligned} \tag{3.2}$$

$$\begin{aligned} \langle \mathbf{v}_{c,x_C} \rangle &= \langle \lambda_a \rangle \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \\ &\quad + (1 - \langle \lambda_a \rangle) \cdot \left((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \end{aligned} \tag{3.3}$$

$$\begin{aligned} \langle \mathbf{v}_{c,x_D} \rangle &= \langle \lambda_a \rangle \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \\ &\quad + (1 - \langle \lambda_a \rangle) \cdot \left((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \right) \end{aligned} \tag{3.4}$$

In order to prove correctness of these equations, we present the truth table of the outputs in Figure 3.10. Observe that all values are correct, as we allow to obtain the negative value of a basic-key.

3.5.1.2 The XOR Gate

We use a similar idea as above to compute the XOR gate. Intuitively, in a XOR gate, there are two cases: $\lambda_a = \lambda_b$ and $\lambda_a \neq \lambda_b$. Multiplying by $\lambda_a - \lambda_b$ gives ± 1 if $\lambda_a \neq \lambda_b$ and 0 if $\lambda_a = \lambda_b$. Furthermore, multiplying by $1 - \lambda_a - \lambda_b$ gives the exact reverse case, it equals 0 if $\lambda_a \neq \lambda_b$

λ_a	λ_b	λ_c	$\langle \mathbf{v}_{c,x_A} \rangle$	$\langle \mathbf{v}_{c,x_B} \rangle$	$\langle \mathbf{v}_{c,x_C} \rangle$	$\langle \mathbf{v}_{c,x_D} \rangle$
0	0	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
0	0	1	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$
0	1	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
0	1	1	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$
1	0	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
1	0	1	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
1	1	0	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
1	1	1	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$

Figure 3.10: The truth table of the vectors for an AND gate computed in Figure 3.8.

and equals ± 1 if $\lambda_a = \lambda_b$. Observe that $\langle \mathbf{v}_{c,x_C} \rangle$ and $\langle \mathbf{v}_{c,x_D} \rangle$ need not be computed at all since $(1-a) \oplus b = a \oplus (1-b)$ and $(1-a) \oplus (1-b) = a \oplus b$. This yields the following equations, where as above, we prove correctness via the truth table given in Figure 3.11.

$$\langle \mathbf{v}_{c,x_A} \rangle = \langle \mathbf{v}_{c,x_D} \rangle = (\langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \right) + (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \tag{3.5}$$

$$\langle \mathbf{v}_{c,x_B} \rangle = \langle \mathbf{v}_{c,x_C} \rangle = (\langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) + (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \right) \tag{3.6}$$

λ_a	λ_b	λ_c	$\langle \mathbf{v}_{c,x_A} \rangle$	$\langle \mathbf{v}_{c,x_B} \rangle$	$\langle \mathbf{v}_{c,x_C} \rangle$	$\langle \mathbf{v}_{c,x_D} \rangle$
0	0	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
0	0	1	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
0	1	0	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$
0	1	1	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$
1	0	0	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
1	0	1	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
1	1	0	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$
1	1	1	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$

Figure 3.11: The truth table of the vectors for a XOR gate computed in Figure 3.8.

3.5.2 Security Of the Modified Protocol

Observe that the only difference in the garbled gates with respect to the depth-4 variant is that the $\langle \mathbf{v}_{c,x_*} \rangle$ values contain the 'tilde' version of the keys. More formally, the $\langle \mathbf{v}_{c,x_*} \rangle$ ciphertexts encrypt the *square root* of the keys, and not the keys themselves. Thus, in the online phase, parties receive the square roots of the keys and need to square them before proceeding. The only issue that needs to be explained here is that the specific square root provided reveals no information. This needs to be justified because if an adversary could know whether $-\tilde{k}$ or $\tilde{k}$ is

computed, then it would know some information about the masks $\lambda_a, \lambda_b, \lambda_c$. However, since the $\tilde{k}$ values are *uniformly distributed* in $\mathbb{F}_p$, and the keys themselves revealed in the offline phase are $k = \tilde{k}^2$, it follows that each of the two square roots of k are equally probable. Stated differently, given k, the distribution over $\tilde{k}$ and $-\tilde{k}$ is identical.

3.5.3 Analysis of Efficiency of the Modified Protocol

As we noted at the beginning of the chapter, the two main sources of overhead that concern the MPC protocols presented are the number of rounds and the number of calls to the ID protocol. The former is not changed by our $\Pi_{\text{Depth}-3}$ variant, but the latter does. To generate the keys in $\Pi_{\text{Preprocessing},3}$, we now perform $2 \cdot W \cdot n^2$ calls to **InputData**, via calls to **RandomElement**. In $\Pi_{\text{Depth}-4}$ we performed $2 \cdot W \cdot n$ calls to generate the keys, so overall we add $2 \cdot W \cdot n \cdot (n-1)$ calls to **InputData**. To analyse the number of homomorphic multiplications we go through each step of the protocol:

- **Generate keys step:** We perform $4 \cdot W \cdot n$ more multiplications (half of them to square the keys, the other half to **Output+** them).
- **Calculate garbled gates step:**
  1. For every AND gate, we used $8 \cdot n + 5$ multiplications in the first variant. Now, by careful rewriting of the equations, we can do this in $12 \cdot n + 8$.
  2. For every XOR gate, we used $6 \cdot n + 3$ multiplications in the first variant. Now we use $8 \cdot n + 4$.
  3. So, on average, we pass from $7 \cdot n + 4$ to $10 \cdot n + 6$ multiplications per gate.

Thus, overall on average we perform $4 \cdot W \cdot n + (3 \cdot n + 2) \cdot G$ more homomorphic multiplications. However, in practice each homomorphic multiplication will be more efficient since the overall depth of the SHE scheme can now be three rather than four.

CHAPTER 4

GARBLING USING OBLIVIOUS TRANSFER

This chapter is based on joint work with Carmit Hazay and Peter Scholl [70], which was presented at ASIACRYPT 2017.

In this chapter, we present a practical, actively secure, constant round multi-party protocol for generating BMR garbled circuits with Free-XOR in the presence of up to $n-1$ out of n corruptions. As in prior constructions, our approach has two phases: a preprocessing phase where the garbled circuit is mutually generated by all parties, and an online phase where the parties obtain the output of the computation. While the online phase is efficient and incurs no extra costs to achieve active security, the focus of this chapter, as well as that of the previous one and other recent works is on optimizing the preprocessing complexity. There, the main bottleneck is with respect to garbling AND gates. In this context, we present two new constant-round protocols for securely generating the garbled circuit:

1. A generic approach using any secret-sharing based MPC protocol for binary circuits, and a correlated oblivious transfer functionality (Section 4.3).

2. A specialized protocol which uses secret-sharing based MPC with information-theoretic MACs, such as TinyOT [58, 104] (Section 4.4). This approach is less general, but requires no additional correlated OTs to compute the garbled circuit.

In both approaches, the cost for garbling an AND gate is essentially *one secure multiplication* on the underlying secret-sharing based protocol, which shows that the gap between constant and non-constant round MPC for boolean circuits is very little in practice.

4.1 Introduction

At the time this work was published, most previous constructions (including the one given in Chapter 3) expressed the garbling function as an arithmetic circuit over a large finite field. For those protocols, garbling even a single AND gate requires computing $O(n)$ multiplications over a large field using Somewhat Homomorphic Encryption (SHE) or MPC. This means they scale at least cubically in the number of parties. In constrast, our protocol only requires one $\mathbb{F}_2$ multiplication per AND gate, so scales with $O(n^2)$. Previous SHE-based protocols also require zero-knowledge proofs of plaintext knowledge of SHE ciphertexts, which in practice are very costly. The MASCOT protocol [80] for secure computation of arithmetic circuits could also be used in [93] instead of SHE, but this still has very high communication costs. We denote by MASCOT-BMR-FX an optimized variant of [93], modified to use Free-XOR as in our protocol, with multiplications in $\mathbb{F}_{2^\kappa}$ done using MASCOT. Finally, a work by Katz et al. [129], concurrent to the one this chapter is based on, relies on an optimized variant of TinyOT and achieves comparable performance results. Table 4.1 shows how the communication complexity of our work compares with other actively secure, constant-round protocols.

Whereas later on [129] also reported implementation results, at the time the work here presented was submitted [70], none of the previous results did so. Our implementation of the TinyOT-based protocol improves upon the best times that would be achievable with SPDZ-BMR and MASCOT by up to *sixty times*. This is because our protocol has lower communication costs than [93] (by at least *two orders of magnitude*) and the main computational costs are from standard symmetric primitives, so far cheaper than using SHE.

Overall, our protocols significantly narrow the gap between the cost of constant-round and many-round MPC protocols for binary circuits. More specifically, this implies that, with current techniques, constant round MPC for binary circuits is not much more expensive than practical, non-constant round protocols. Additionally, both of our protocols have potential for future improvement by optimizing existing non-constant round protocols: a practical implementation of MiniMAC [49] would lead to a very efficient approach with our generic protocol, whilst any future improvements to multi-party TinyOT would directly give a similar improvement to our second protocol.

4.1.1 Our Contributions

Our constructions employ several very appealing features. For a start, we embed into the modelling of the preprocessing functionality, which computes the garbled circuit, an additive error introduced into the garbling by the adversary. Concretely, we extend the functionality from [93] so that it obtains a vector of additive errors from the adversary to be applied to each garbled gate, which captures the fact that the adversary may submit inconsistent keys and pseudorandom function (PRF) values. We further strengthen this by allowing the adversary to pick the error

Protocol	Based on	Free XOR	Comms. per Garbled Gate
SPDZ-BMR [93]	SHE + ZKPoPK	✗	$O(n^4\kappa)$
SHE-BMR §3	SHE (depth 4) + ZKPoPK	✗	$O(n^3\kappa)$
MASCOT-BMR-FX	OT	✓	$O(n^3\kappa^2)$
This work §4.3	OT + [75]	✓	$O(n^2\kappa + \mathsf{poly}(n))$
This work §4.4	TinyOT	✓	$O(n^2B^2\kappa)$
[129] (concurrent)	Optimized TinyOT	✓	$O(n^2B\kappa)$

Table 4.1: Comparison of actively secure, constant round MPC protocols. $B = O(1 + s/\log|C|)$ is a cut-and-choose parameter, which in practice is between 3–5. Our second protocol can also be based upon optimized TinyOT to obtain the same complexity as [129].

adaptively after seeing the garbled circuit (in prior constructions this error is independent of the garbling) and allowing corrupt parties to choose their own PRF keys, possibly not at random. This requires a new analysis and proof of the online phase.

Secondly, we devise a new consistency check to enforce correctness of inputs to correlated OT, which is based on very efficient linear operations similar to recent advances in homomorphic commitments [37]. This check, combined with our improved error analysis for the online phase, allows the garbled circuit to be created without authenticating any of the parties' keys or PRF values, which removes a significant cost from previous works (saving a factor of $\Omega(n)$).

GENERIC CONSTRUCTION (SECTION 4.3). In the first, more general method, every pair of parties needs to run one correlated OT per AND gate, which costs $O(\kappa)$ communication for security parameter κ. Combining this with the overhead induced by the correlated OTs in our protocol, we obtain total complexity $O(|C|\kappa n^2)$, assuming only symmetric primitives and $O(\kappa)$ seed OTs between every pair of parties. This gives an overall communication cost of $O(M + |C|\kappa n^2)$ to evaluate a circuit C, where M is the cost of evaluating $|C|$ AND gates in the secret-sharing based protocol, Π. To realize Π, we can define a functionality with multiplication depth 1 that computes all the AND gates in parallel (these multiplications can be computed in parallel as they are independent of the parties' inputs). Furthermore, the [75] compiler can be instantiated with semi-honest [64] as the inner protocol and [43] as the outer protocol. By Theorem 2, Section 5 from [75], for some constant number of parties $m \geq 2$, the functionality can be computed with communication complexity $O(|C|)$ plus low order terms that depend on a statistical parameter s, the circuit's depth and $\log|C|$. As in [75], this extends to the case of a non-constant number of parties n, in which case the communication complexity grows by an additional factor of $|C|\mathsf{poly}(n)$.

Another candidate for instantiating Π would be to use an MPC protocol optimized for SIMD binary circuits such as MiniMAC [49]. This is because in our construction, all the AND gates can be computed in parallel. Currently, the only known preprocessing methods [58]

for MiniMAC are not practical, but this seems to be an interesting direction to explore.

TINYOT-BASED CONSTRUCTION (SECTION 4.4). TinyOT is currently the most practical approach to secret-sharing based MPC on binary circuits, so the second method leads to a highly practical protocol for constant-round secure computation. The complexity is essentially the same as TinyOT, as here we do not require any additional OTs. However, the protocol is less general and has worse asymptotic communication complexity, since TinyOT costs either $O(|C|B\kappa n^2)$ (with 2 parties or the recent protocol of [128]), or $O(|C|B^2\kappa n^2)$ (with [58]), where $B = O(1 + s/\log|C|)$ (and in practice is between 3–5), and s is the statistical security parameter.

The results of our work, particularly regarding our modelling of circuit-garbling, have already proved to be useful for other authors as explicitly mentioned in [27, 67]. Remarkably, those works had very different goals: Whereas [27] improves the efficiency of BMR-style protocols in the honest majority setting, [67] provides the first round-optimal actively secure MPC protocol under standard assumptions.

4.1.1.1 Implementation.

We demonstrate the practicality of our TinyOT-based protocol with an implementation, and perform experiments with up to 9 parties securely computing the AES and SHA-256 circuits. In a 1Gbps LAN setting, we can securely compute the AES circuit with 9 parties in just 620ms. This improves upon the best possible performance that would be attainable using [93] by around 60 times. The details of our implementation can be found in Section 4.6.

4.1.1.2 Concurrent work

Two works by Wang, Ranellucci and Katz, concurrent to the one described in this chapter, introduced constant round two-party [128] and multi-party [129] protocols based on *authenticated garbling*, in which the preprocessing phase is based on TinyOT. The less general part of our work is conceptually quite similar, since both involve generating a garbled circuit in a distributed manner using TinyOT. The main difference seems to be that our protocol is symmetric, since all parties evaluate the same garbled circuit. With authenticated garbling, the garbled circuit is only evaluated by one party. This makes the garbled circuit slightly smaller, since there are $n-1$ sets of keys instead of n, but the online phase requires at least one more round of interaction (if all parties learn the output). The works of Wang et al. also contain concrete and asymptotic improvements to the two-party and multi-party TinyOT protocols, which improve upon our protocol in Section 4.7 by a factor of $O(s/\log|C|)$ where s is a statistical parameter. These improvements can be directly plugged into our second garbling protocol. We remark that the two-party protocol in [128] inspired our use of TinyOT MACs to perform the bit/string multiplications in our protocol from Section 4.4. The rest of our work is independent. Another difference is that

our protocol from Section 4.3 is more generic, since $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ can be implemented with *any* secret-sharing based bit-MPC protocol, rather than just TinyOT. This can be instantiated with [75] to obtain a constant-round protocol with complexity $O(|C|(\kappa n^2 + \text{poly}(n)))$ in the OT-hybrid model. The multi-party paper [129] does not have an analogous generic result.

4.1.2 Technical Overview

Our protocol is based on the recent Free-XOR variant of BMR garbling used for semi-honest MPC in [25]. In that scheme, a garbling of the g-th AND gate with input wires u, v and output wire w, consists of the $4n$ values (where n is the number of parties):

$$\begin{aligned}\tilde{g}^j_{a,b} = &\left(\bigoplus_{i=1}^{n} F_{k^i_{u,a},k^i_{v,b}}(g\|j)\right) \oplus k^j_{w,0} \\ &\oplus \left(\boldsymbol{r}^j((\lambda_u \oplus a)(\lambda_v \oplus b) \oplus \lambda_w)\right), \quad (a,b) \in \{0,1\}^2,\ j \in [n]\end{aligned} \tag{4.1}$$

Here, F is a double-key PRF[1], $\boldsymbol{r}^j \in \{0,1\}^\kappa$ is a fixed correlation string for Free-XOR known to party P_j, and the keys $k^j_{u,a}, k^j_{v,b} \in \{0,1\}^\kappa$ are also known to P_j. Furthermore, the wire masks $\lambda_u, \lambda_v, \lambda_w \in \{0,1\}$ are random, additively secret-shared bits known by no single party.

Let us give an overview of BMR, as described in Section 2.6, when the 'vector double encryption' is instantiated using PRFs as in equation (4.1). The main idea behind BMR in this case is to compute the garbling, except for the PRF values, with a general MPC protocol. The analysis of [93] showed that it is not necessary to prove in zero-knowledge that every party inputs the correct PRF values to the MPC protocol that computes the garbling. This is because when evaluating the garbled circuit, each party P_j can check that the decryption of the j-th entry in every garbled gate gives one of the keys $k^j_{w,0}, k^j_{w,1}$ and this check would overwhelmingly fail if any PRF value was incorrect. It further implies that the adversary cannot flip the value transmitted through some wire as that would require from it to guess a key.

Our garbling protocol proceeds by computing a random, *unauthenticated*, additive secret sharing of the garbled circuit. This differs from previous works which obtain *authenticated* sharings of the entire garbled circuit (e.g. [93] uses MACs and Chapter 3 employs SHE ciphertexts). Our protocol greatly reduces this complexity, since the PRF values and keys (on the first line of equation (4.1)) do not need to be authenticated. The main challenge, therefore, is to compute shares of the products on the second line of (4.1). Similarly to [25], an important observation that improves efficiency is the fact that these multiplications are either between two secret-shared bits, or a secret-shared bit and a fixed, secret string. Thus, we do not need the full power of an

[1]Actually, as enabling Free-XOR causes all the keys pairs for any wire to be correlated, we need a more advanced notion which we present in Section 4.2.2.

MPC protocol for arithmetic circuit evaluation over $\mathbb{F}_{2^\kappa}$ or $\mathbb{F}_p$ (for large p), as used in previous works.

To compute the bit product $\lambda_u \cdot \lambda_v$, we can use any actively secure GMW-style MPC protocol for binary circuits. This protocol is only needed for computing *one secure AND per garbled AND gate*, since all bit products in $\tilde{g}^j_{a,b}$ can be computed as linear combinations of $\lambda_u \cdot \lambda_v$, λ_u and λ_v. We then need to multiply the resulting secret-shared bits by the string $\boldsymbol{r}^j$, known to P_j. We give two variants for computing this product, the first one being more general and the second more concretely efficient. In more details,

1. The first solution performs the multiplication by running actively secure correlated OT between P_j and every other party, where P_j inputs $\boldsymbol{r}^j$ as the fixed OT correlation. The parties then run a consistency check by applying a universal linear hash function to the outputs and sacrificing a few OTs, ensuring the correct inputs were provided to the OT. This protocol is presented in Section 4.3.

2. The second method requires using a 'TinyOT'-style protocol [34, 58] based on information-theoretic MACs, and allows us to compute the bit/string products directly from the MACs, provided each party's MAC key is chosen to be the same string $\boldsymbol{r}^j$ used in the garbling. This saves interaction since we do not need any additional OTs. This protocol is presented in Section 4.4.

After creating shares of all these products, the parties can compute shares of the whole garbled circuit. These shares must then be rerandomized, before they can be broadcast. Opening the garbled circuit in this way allows a corrupt party to introduce further errors into the garbling by changing their share, even *after learning the correct garbled circuit*, since we may have a rushing adversary. Nevertheless, we prove that the BMR online phase remains secure when this type of error is allowed, as it would only lead to an abort. This significantly strengthens the result from [93], which only allowed corrupt parties to provide incorrect PRF values, and is an important factor that allows our preprocessing protocol to be so efficient.

4.2 Preliminaries

For vectors $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ and $\boldsymbol{y} \in \mathbb{F}_2^m$, the *tensor product* (or *outer product*) $\boldsymbol{x} \otimes \boldsymbol{y}$ is defined as the $n \times m$ matrix over $\mathbb{F}_2$ where the i-th row is $x_i \cdot \boldsymbol{y}$. We use the following property.

Fact 4.2.1. *If $\boldsymbol{x} \in \mathbb{F}_2^n, \boldsymbol{y} \in \mathbb{F}_2^m$ and $\boldsymbol{M} \in \mathbb{F}_2^{m \times n}$ then*

$$\boldsymbol{M} \cdot (\boldsymbol{x} \otimes \boldsymbol{y}) = (\boldsymbol{M} \cdot \boldsymbol{x}) \otimes \boldsymbol{y}.$$

4.2.1 Security and Communication Models

We prove security of our protocols in the Universal Composability framework (see Section 2.4). The adversary model we consider is a static, active adversary who corrupts up to $n-1$ out of n parties. We recall this means that the identities of the corrupted parties are fixed at the beginning of the protocol, and they may deviate arbitrarily from the protocol.

We assume all parties are connected via authenticated communication channels, as well as secure point-to-point channels and a broadcast channel. The default method of communication in the protocols in this chapter is authenticated channels, unless otherwise specified. Note that in practice, these can all be implemented with standard techniques (in particular, for broadcast a simple 2-round protocol suffices, since we allow abort [65]).

4.2.2 Circular 2-Correlation Robust Pseudorandom Functions

The BMR garbling technique from [93] is proven secure based on a pseudorandom function (PRF) with multiple keys. When the keys are independently chosen then security with multiple keys can be derived from the standard PRF notion (defined in Section 2.2.2) by a simple hybrid argument. However, since our scheme supports Free-XOR, we require a stronger assumption, discussed next.

We adapt the definition of correlation robustness with circularity from [40] given for hash functions to double-key PRFs. This definition captures the related key and circularity requirements induced by supporting the Free-XOR technique. Formally, fix some function $F : \{0,1\}^n \times \{0,1\}^\kappa \times \{0,1\}^\kappa \mapsto \{0,1\}^\kappa$. Let b_1, b_2, b_3 take values in $\{0,1\}$, and $k_1, k_2, \boldsymbol{r}$ in $\{0,1\}^\kappa$. We define an oracle $\mathsf{Circ}_{\boldsymbol{r}}$ as follows:

- $\mathsf{Circ}_{\boldsymbol{r}}(k_1, k_2, g, j, b_1, b_2, b_3)$ outputs $F_{k_1 \oplus b_1\boldsymbol{r}, k_2 \oplus b_2\boldsymbol{r}}(g \| j) \oplus b_3\boldsymbol{r}$.

The outcome of oracle Circ is compared with the a random string of the same length computed by an oracle Rand:

- $\mathsf{Rand}(k_1, k_2, g, j, b_1, b_2, b_3)$: if this input was queried before then return the answer given previously. Otherwise choose $u \leftarrow \{0,1\}^\kappa$ and return u.

Definition 4.1 (Circular 2-correlation robust PRF)**.** A PRF F is *circular 2-correlation robust* if for any non-uniform polynomial-time distinguisher $\mathscr{D}$ making legal queries to its oracle, there exists a negligible function negl such that:

$$\left|\Pr[\boldsymbol{r} \leftarrow \{0,1\}^\kappa; \mathscr{D}^{\mathsf{Circ}_{\boldsymbol{r}}(\cdot)}(1^\kappa) = 1] - \Pr[\mathscr{D}^{\mathsf{Rand}(\cdot)}(1^\kappa) = 1]\right| \leq \mathsf{negl}(\kappa).$$

As in [40], some trivial queries must be ruled out. Specifically, the distinguisher is restricted as follows: (1) it is not allowed to make any query of the form $\mathscr{O}(k_1, k_2, g, j, 0, 0, b_3)$ (since it can compute $F_{k_1,k_2}(g \| j)$ on its own) and (2) it is not allowed to query both tuples $\mathscr{O}(k_1, k_2, g, j, b_1, b_2, 0)$

and $\mathcal{O}(k_1, k_2, g, j, b_1, b_2, 1)$ for any values k_1, k_2, g, j, b_1, b_2 (since that would allow it to trivially recover the global difference). We say that any distinguisher respecting these restrictions makes legal queries.

4.2.3 Almost-1-Universal Linear Hashing

We use a family of almost-1-universal linear hash functions over $\mathbb{F}_2$, defined by:

Definition 4.2 (Almost-1-Universal Linear Hashing)**.** We say that a family $\mathscr{H}$ of linear functions $\mathbb{F}_2^m \to \mathbb{F}_2^s$ is ε-almost 1-universal, if it holds that for every non-zero $\boldsymbol{x} \in \mathbb{F}_2^m$ and for every $\boldsymbol{y} \in \mathbb{F}_2^s$:

$$\Pr_{\boldsymbol{H} \leftarrow \mathscr{H}}[\boldsymbol{H}(\boldsymbol{x}) = \boldsymbol{y}] \leq \varepsilon$$

where $\boldsymbol{H}$ is chosen uniformly at random from the family $\mathscr{H}$. We will identify functions $\boldsymbol{H} \in \mathscr{H}$ with their $s \times m$ transformation matrix, and write $\boldsymbol{H}(\boldsymbol{x}) = \boldsymbol{H} \cdot \boldsymbol{x}$.

This definition is slightly stronger than a family of almost-universal linear hash functions (where the above need only hold for $\boldsymbol{y} = 0$, as in [37]). However, this is still much weaker than *2-universality* (or *pairwise independence*), which a linear family of hash functions cannot achieve, because $\boldsymbol{H}(0) = 0$ always. The two main properties affecting the efficiency of a family of hash functions are the *seed size*, which refers to the length of the description of a random function $\boldsymbol{H} \leftarrow \mathscr{H}$, and the *computational complexity* of evaluating the function. The simplest family of almost-1-universal hash functions is the set of all $s \times m$ matrices; however, this is not efficient as the seed size and complexity are both $O(m \cdot s)$. Recently, in [37], it was shown how to construct a family with seed size $O(s)$ and complexity $O(m)$, which is asymptotically optimal. A more practical construction is a polynomial hash based on GMAC (used also in [107]), described as follows (here we assume that s divides m, for simplicity):

- Sample a random seed $\alpha \leftarrow \mathbb{F}_{2^s}$
- Define $\boldsymbol{H}_\alpha$ to be the function:

  $$\boldsymbol{H}_\alpha : \mathbb{F}_{2^s}^{m/s} \to \mathbb{F}_{2^s}, \quad \boldsymbol{H}_\alpha(x_1, \ldots, x_{m/s}) = \alpha \cdot x_1 + \alpha^2 \cdot x_2 + \cdots + \alpha^{m/s} \cdot x_{m/s}$$

  Note that by viewing elements of $\mathbb{F}_{2^s}$ as vectors in $\mathbb{F}_2^s$, multiplication by a fixed field element $\alpha^i \in \mathbb{F}_{2^s}$ is linear over $\mathbb{F}_2$. Therefore, $\boldsymbol{H}_\alpha$ can be seen as an $\mathbb{F}_2$-linear map, represented by a unique matrix in $\mathbb{F}_2^{s \times m}$.

Here, the seed is short, but the computational complexity is $O(m \cdot s)$. However, in practice when $s = 128$ the finite field multiplications can be performed very efficiently in hardware on modern CPUs. Note that this gives a 1-universal family with $\varepsilon = \frac{m}{s} \cdot 2^{-s}$. This can be improved to 2^{-s} (i.e. perfect), at the cost of a larger seed, by using m/s distinct elements α_i, instead of powers of α.

4.2.4 Commitment Functionality

We use a commitment functionality $\mathcal{F}_{\text{Commit}}$ (Figure 4.1). This can be implemented in the random oracle model by defining $\texttt{Commit}(x, P_i) = \mathsf{H}(x, i, r)$, where H is a random oracle and $r \leftarrow \{0,1\}^\kappa$.

Functionality $\mathcal{F}_{\text{Commit}}$

Commit: On input $(\texttt{Commit}, x, i, \tau_x)$ from P_i, store (x, i, τ_x) and output (i, τ_x) to all parties.

Open: On input $(\texttt{Open}, i, \tau_x)$ by P_i, output (x, i, τ_x) to all parties.

If instead $(\texttt{NoOpen}, i, \tau_x)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_x)$ to all parties.

Figure 4.1: Commitments functionality.

4.2.5 Coin-Tossing Functionality

We also require a standard coin-tossing functionality, $\mathcal{F}_{\text{Rand}}$ (Figure 4.2), which can be implemented using commitments ($\mathcal{F}_{\text{Commit}}$ above) to random values.

Functionality $\mathcal{F}_{\text{Rand}}$

Upon receiving $(\texttt{rand}, S)$ from all parties, where S is any efficiently sampleable set, sample $r \leftarrow S$, send r to $\mathscr{A}$ and wait for its input. If $\mathscr{A}$ inputs $\texttt{OK}$ then output r to all parties, otherwise output $\perp$.

Figure 4.2: Coin-tossing functionality.

4.2.6 Correlated Oblivious Transfer

In this work we use an actively secure protocol for oblivious transfer (OT) on correlated pairs of strings of the form $(a_i, a_i \oplus \Delta)$, where Δ is fixed for every OT. The TinyOT protocol [104] for secure two-party computation constructs such a protocol, and a significantly optimized version of this is given in [107]. The communication cost is roughly $\kappa + s$ bits per OT. The ideal functionality is shown in Figure 4.3.

Fixed Correlation OT Functionality $\mathcal{F}_{\Delta\text{-ROT}}$

Initialize: Upon receiving $(\texttt{init}, \Delta)$, where $\Delta \in \{0,1\}^\kappa$ from P_S and $(\texttt{init})$ from P_R, store Δ. Ignore any subsequent initcommands.

Extend: Upon receiving $(\texttt{extend}, x_1, \ldots, x_m)$ from P_R, where $x_i \in \{0,1\}$, and $(\texttt{extend})$ from P_S, do the following:

- Sample $t_i \in \{0,1\}^\kappa$, for $i \in [m]$. If P_R is corrupted then wait for $\mathscr{A}$ to input t_i.
- Compute $q_i = t_i + x_i \cdot \Delta$, for $i \in [m]$.
- If P_S is corrupted then wait for $\mathscr{A}$ to input $q_i \in \{0,1\}^\kappa$ and recompute $t_i = q_i + x_i \cdot \Delta$.
- Output t_i to P_R and q_i to P_S, for $i \in [m]$.

Figure 4.3: Fixed correlation oblivious transfer functionality.

The Bit MPC Functionality - $\mathcal{F}_{\text{Bit}\times\text{Bit}}$

The functionality runs with parties $P_1,\ldots,P_n$ and an adversary $\mathcal{A}$. The functionality maintains a dictionary, $\texttt{Val} \leftarrow \{\}$, to keep track of values in $\mathbb{F}_2$.

Input: On receiving $(\texttt{Input}, \texttt{id}_1,\ldots,\texttt{id}_\ell, x_1,\ldots,x_\ell, P_j)$ from party P_j and $(\texttt{Input}, \texttt{id}_1,\ldots,\texttt{id}_\ell, P_j)$ from all other parties, where $x_i \in \mathbb{F}_2$, store $\texttt{Val}[\texttt{id}_i] \leftarrow x_i$ for $i \in [\ell]$.

Add: On input $(\text{Add}, \overline{\texttt{id}}, \texttt{id}_1,\ldots,\texttt{id}_\ell)$ from all parties, where $(\texttt{id}_1,\ldots,\texttt{id}_\ell)$ are keys in $\texttt{Val}$, set $\texttt{Val}[\overline{\texttt{id}}] \leftarrow \sum_{i=1}^{\ell} \texttt{Val}[\texttt{id}_i]$.

Multiply: On input $(\texttt{multiply}, \overline{\texttt{id}}, \texttt{id}_1, \texttt{id}_2)$ from all parties, where $(\texttt{id}_1, \texttt{id}_2)$ are keys in $\texttt{Val}$, compute $y \leftarrow \texttt{Val}[\texttt{id}_1] \cdot \texttt{Val}[\texttt{id}_2]$. Receive shares $y^i \in \mathbb{F}_2$ from $\mathcal{A}$, for $i \in I$, then sample random honest parties' shares $y^j \in \mathbb{F}_2$, for $j \notin I$, such that $\sum_{i=1}^{n} y^i = y$. Send y^i to party P_i, for $i \in [n]$, and store the value $\texttt{Val}[\overline{\texttt{id}}] \leftarrow y$.

Open: On input $(\text{Open}, \texttt{id})$ from all parties, where $\texttt{id}$ is a key in $\texttt{Val}$, send $x \leftarrow \texttt{Val}[\texttt{id}]$ to $\mathcal{A}$. Wait for an input from $\mathcal{A}$. If it inputs $\texttt{OK}$ then output x to all parties, otherwise output $\perp$ and terminate.

Figure 4.4: Functionality for GMW-style MPC for binary circuits.

4.2.7 Functionality for Secret-Sharing-Based MPC

We make use of a general, actively secure protocol for secret-sharing-based MPC for binary circuits, which is modeled by the functionality $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ in Figure 4.4. This functionality allows parties to provide private inputs, which are then stored and can be added or multiplied internally by $\mathcal{F}_{\text{Bit}\times\text{Bit}}$, and revealed if desired. Note that we also need the **Multiply** command to output a random additive secret-sharing of the product to all parties, which is the main difference with the generic MPC functionality given in Figure 2.1 and essentially assumes that the underlying protocol is based on secret-sharing.

We use the notation $\langle x \rangle$ to represent a secret-shared value x that is stored internally by $\mathcal{F}_{\text{Bit}\times\text{Bit}}$, and define x^i to be party P_i's additive share of x (if it is known). We also define the $+$ and $\cdot$ operators on two shared values $\langle x \rangle, \langle y \rangle$ to call the **Add** and **Multiply** commands of $\mathcal{F}_{\text{Bit}\times\text{Bit}}$, respectively, and return the identifier associated with the result.

4.3 Generic Protocol for Multi-Party Garbling

We now describe our generic method for creating the garbled circuit using any secret-sharing based MPC protocol (modeled by $\mathcal{F}_{\text{Bit}\times\text{Bit}}$) and the correlated OT functionality $\mathcal{F}_{\Delta\text{-ROT}}$. We first describe the functionality in Section 4.3.1 and the protocol in Section 4.3.2, and then analyse its security in Sections 4.3.4–4.3.5.

4.3.1 The Preprocessing Functionality

The preprocessing functionality, formalized in Figure 4.5, captures the generation of the garbled circuit as well as an error introduced by the adversary. The adversary is allowed to submit

an additive error, chosen adaptively after seeing the garbled circuit, which is added by the functionality to each entry when the garbled circuit is opened.

4.3.2 Protocol Overview

The garbling protocol, shown in Figure 4.6, proceeds in three main stages. Firstly, the parties locally sample all of their keys and shares of wire masks for the garbled circuit. Secondly, the parties compute shares of the products of the wire masks and each party's global difference string; these are then used by each party to locally obtain a share of the entire garbled circuit. Finally, the bit masks for the output wires are opened to all parties. The opening of the garbled circuit is shown in Figure 4.7.

Concretely, each party P_i starts by sampling a global difference string $\boldsymbol{r}^i \leftarrow \{0,1\}^\kappa$, and for each wire w which is an output wire of an AND gate, or an input wire, P_i also samples the keys $k^i_{w,0}$, $k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i$ and an additive share of the wire mask, $\lambda^i_w \leftarrow \mathbb{F}_2$. As in [25], we let P_i input the actual wire mask (instead of a share) for every input wire associated with P_i's input.

In step 4.3.2, the parties compute additive shares of the bit products $\lambda_{uv} = \lambda_u \cdot \lambda_v \in \mathbb{F}_2$, and then, for each $j \in [n]$, shares of:

$$\lambda_u \cdot \boldsymbol{r}^j, \quad \lambda_v \cdot \boldsymbol{r}^j, \quad \lambda_{uvw} \cdot \boldsymbol{r}^j \in \mathbb{F}_2^\kappa \tag{4.2}$$

where $\lambda_{uvw} := \lambda_{uv} \oplus \lambda_w$, and u, v and w are the input and output wires of AND gate g. We note that (as observed in [25]) only one bit/bit product and $3n$ bit/string products are necessary, even though each gate has $4n$ entries, due to correlations between the entries, as can be seen below.

We compute the bit multiplications using the $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ functionality on the bits that are already stored by $\mathscr{F}_{\text{Bit}\times\text{Bit}}$. To compute the bit/string multiplications in equation (4.2), we use correlated OT, followed by a consistency check to verify that the parties provided the correct shares of λ_w and correlation $\boldsymbol{r}^i$ to each $\mathscr{F}_{\Delta\text{-ROT}}$ instance; see Section 4.3.3 for details.

Using shares of the bit/string products, the parties can locally compute an unauthenticated additive share of the entire garbled circuit (steps 3d–4). First, for each of the four values $(a,b) \in \{0,1\}^2$, each party $P_i, i \neq j$ computes the share

$$\rho^i_{j,a,b} = \begin{cases} a \cdot (\lambda_v \cdot \boldsymbol{r}^j)^i \oplus b \cdot (\lambda_u \cdot \boldsymbol{r}^j)^i \oplus (\lambda_{uvw} \cdot R^j)^i & \text{if } i \neq j \\ a \cdot (\lambda_v \cdot \boldsymbol{r}^j)^i \oplus b \cdot (\lambda_u \cdot \boldsymbol{r}^j)^i \oplus (\lambda_{uvw} \cdot R^j)^i \oplus a \cdot b \cdot \boldsymbol{r}^j & \text{if } i = j \end{cases}$$

These define additive shares of the values

$$\begin{aligned} \rho_{j,a,b} &= R^j \cdot (a \cdot \lambda_v \oplus b \cdot \lambda_u \oplus \lambda_{uvw} \oplus a \cdot b) \\ &= \boldsymbol{r}^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \end{aligned}$$

Each party's share of the garbled circuit is then obtained by adding the appropriate PRF values and keys to the shares of each $\rho_{j,a,b}$. To conclude the **Garbling** stage, the parties reveal

Functionality $\mathcal{F}_{\text{Preprocessing}}$

Let F be a circular 2-correlation robust PRF. The functionality runs with parties $P_1,\ldots,P_n$ and an adversary $\mathcal{A}$, who corrupts a subset $I \subset [n]$ of parties.

Garbling: On input $(\texttt{Garbling}, C_f)$ from all parties, where C_f is a boolean circuit, denote by W its set of wires and by G its set of AND gates. The functionality is defined as follows:

- Sample a global difference $\boldsymbol{r}^j \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receive corrupt parties' strings $\boldsymbol{r}^i \in \{0,1\}^\kappa$ from $\mathcal{A}$, for $i \in I$.
- Passing topologically through all the wires $w \in W$ of the circuit:
  - If w is an input wire:
    1. Sample $\lambda_w \leftarrow \{0,1\}$. If P_j, the party who provides input on that wire in the online phase, is corrupt, instead receive λ_w from $\mathcal{A}$.
    2. Sample a key $k^j_{w,0} \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receive corrupt parties' keys $k^i_{w,0}$ from $\mathcal{A}$, for $i \in I$. Define $k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i$ for all $i \in [n]$.
  - If w is the output of an AND gate:
    1. Sample $\lambda_w \leftarrow \{0,1\}$.
    2. Sample a key $k^j_{w,0} \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receive corrupt parties' keys $k^i_{w,0}$ from $\mathcal{A}$, for $i \in I$. Set $k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i$, for $i \in [n]$.
  - If w is the output of a XOR gate, and u and v its input wires:
    1. Compute and store $\lambda_w = \lambda_u \oplus \lambda_v$.
    2. For $i \in [n]$, set $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i$.
- For every AND gate $g \in G$, the functionality computes the $4n$ entries of the garbled version of g as:

$$\tilde{g}^j_{a,b} = \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a},k^i_{v,b}}(g\|j)\right) \oplus k^j_{w,0} \oplus \left(\boldsymbol{r}^j \cdot ((\lambda_u \oplus a)\cdot(\lambda_v \oplus b) \oplus \lambda_w)\right), \quad (a,b) \in \{0,1\}^2,\ j \in [n].$$

  Set $\tilde{g}_{a,b} = \tilde{g}^1_{a,b} \circ \ldots \circ \tilde{g}^n_{a,b} \quad (a,b) \in \{0,1\}^2$. The functionality stores the values $\tilde{g}_{a,b}$.
- Wait for an input from $\mathcal{A}$. If it inputs $\texttt{OK}$ then output λ_w to all parties for each circuit-output wire w, and output to each P_i all the keys $\{k^i_{w,0}\}_{w\in W}$, and $\boldsymbol{r}^i$. Otherwise, output $\perp$ and terminate.

Open Garbling: On receiving $(\texttt{OpenGarbling})$ from all parties, when the **Garbling** command has already run successfully, the functionality sends to $\mathcal{A}$ the values $\tilde{g}_{a,b}$ for all $g \in G$ and waits for a reply.

- If $\mathcal{A}$ returns $\perp$ then the functionality aborts.
- Otherwise, the functionality receives $\texttt{OK}$ and an additive error $\boldsymbol{e} = \{e^{a,b}_g\}_{a,b\in\{0,1\},g\in G}$ chosen by $\mathcal{A}$. Afterwards, it sends to all parties the garbled circuit $\tilde{g}_{a,b} \oplus e^{a,b}_g$ for all $g \in G$ and $a,b \in \{0,1\}$.

Figure 4.5: The Preprocessing Functionality $\mathcal{F}_{\text{Preprocessing}}$.

The Preprocessing Protocol – $\Pi_{\text{Preprocessing}}$

Given a gate g, we denote by u (resp. v) its left (resp. right) input wire, and by w its output wire. $\langle\cdot\rangle^i$ denotes the i-th share of an authenticated bit and $(\cdot)^i$ the i-th share of a string. Let $F : \{0,1\}^{2\kappa} \times [|G|] \times [n] \to \{0,1\}^\kappa$ be a circular 2-correlation robust PRF, and $\mathscr{G} : \{0,1\}^\kappa \to \{0,1\}^{4n\kappa|G|}$ be a PRG.

Garbling: 1. Each party P_i samples a random key offset $\boldsymbol{r}^i \leftarrow \mathbb{F}_2^\kappa$.

2. **Generate wire masks and keys:** Passing through the wires of the circuit topologically:
   - If w is a *circuit-input* wire, and P_j is the party whose input is associated with it:
     a) P_j calls **Input** on $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ with a randomly sampled $\lambda_w \in \{0,1\}$ to obtain $\langle\lambda_w\rangle$. P_j defines the share $\lambda_w^j = \lambda_w$, every other P_i sets $\lambda_w^i = 0$.
     b) Every P_i samples a key $k_{w,0}^i \leftarrow \{0,1\}^\kappa$ and sets $k_{w,1}^i = k_{w,0}^i \oplus \boldsymbol{r}^i$.
   - If the wire w is the output of an AND gate:
     a) Each P_i calls **Input** on $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ with a randomly sampled $\lambda_w^i \leftarrow \{0,1\}$. The parties then compute the secret-shared wire mask as $\langle\lambda_w\rangle = \sum_{i\in[n]}\langle\lambda_w^i\rangle$.
     b) Every P_i samples a key $k_{w,0}^i \leftarrow \{0,1\}^\kappa$ and sets $k_{w,1}^i = k_{w,0}^i \oplus \boldsymbol{r}^i$.
   - If the wire w is the output of a XOR gate:
     a) The parties compute the mask on the output wire as $\langle\lambda_w\rangle = \langle\lambda_u\rangle + \langle\lambda_v\rangle$.
     b) Every P_i sets $k_{w,0}^i = k_{u,0}^i \oplus k_{v,0}^i$ and $k_{w,1}^i = k_{w,0}^i \oplus \boldsymbol{r}^i$.

3. **Secure product computations:**
   a) For each AND gate $g \in G$, the parties compute $\langle\lambda_{uv}\rangle = \langle\lambda_u\rangle \cdot \langle\lambda_v\rangle$ by calling **Multiply** on $\mathscr{F}_{\text{Bit}\times\text{Bit}}$.
   b) Each P_i calls **Input** on $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ with randomly sampled bits $\hat{x}_1^i, \ldots, \hat{x}_s^i$. For $\ell \in [s]$, the parties compute secret-shared mask $\langle\hat{x}_\ell\rangle = \sum_{i\in[n]}\langle\hat{x}_\ell^i\rangle$.
   c) For every $j \in [n]$, the parties run the subprotocol $\Pi_{\text{Bit}\times\text{String}}$, where P_j inputs $\boldsymbol{r}^j$ and everyone inputs the $3|G| + s$ shared bits:
   $$(\langle\lambda_u\rangle, \langle\lambda_v\rangle, \langle\lambda_{uv}\rangle + \langle\lambda_w\rangle)_{(u,v,w)} \quad \text{and} \quad (\langle\hat{x}_1\rangle, \ldots, \langle\hat{x}_s\rangle).$$
   where the (u,v,w) indices are taken over the input/output wires of each AND gate $g \in G$.
   d) For each AND gate $g \in G$, party P_i obtains from $\Pi_{\text{Bit}\times\text{String}}$ an additive share of the $3n$ values (each defined as one row of the matrix $\boldsymbol{Z}_j$ in this subprotocol):
   $$\lambda_u \cdot \boldsymbol{r}^j, \quad \lambda_v \cdot \boldsymbol{r}^j, \quad \lambda_{uvw} \cdot \boldsymbol{r}^j, \qquad \text{for } j \in [n]$$
   where $\lambda_{uvw} := \lambda_{uv} + \lambda_w$. Each P_i then uses these to compute a share of
   $$\rho_{j,a,b} = \lambda_{uvw} \cdot \boldsymbol{r}^j \oplus a \cdot \lambda_v \cdot \boldsymbol{r}^j \oplus b \cdot \lambda_u \cdot \boldsymbol{r}^j \oplus a \cdot b \cdot \boldsymbol{r}^j$$

4. **Garble gates:** For each AND gate $g \in G$, each $j \in [n]$, and the four combinations of $a, b \in \{0,1\}^2$, the parties compute shares of the j-th entry of the garbled gate $\tilde{g}_{a,b}$:
   - P_j sets $(\tilde{g}_{a,b}^j)^j = \rho_{j,a,b}^j \oplus F_{k_{u,a}^j, k_{v,b}^j}(g\|j) \oplus k_{w,0}^j$.
   - For every $i \neq j$, P_i sets $(\tilde{g}_{a,b}^j)^i = \rho_{j,a,b}^i \oplus F_{k_{u,a}^i, k_{v,b}^i}(g\|j)$.

5. **Reveal masks for output wires:** For every circuit-output-wire w, the parties call **Open** on $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ to reveal λ_w to all the parties.

Figure 4.6: The preprocessing protocol that realizes $\mathscr{F}_{\text{Preprocessing}}$ in the $\{\mathscr{F}_{\Delta\text{-ROT}}, \mathscr{F}_{\text{Bit}\times\text{Bit}}, \mathscr{F}_{\text{Rand}}\, \mathscr{F}_{\text{Commit}}\}$-hybrid model.

The Preprocessing Protocol $\Pi_{\text{Preprocessing}}$ – Open Garbling Stage

Open Garbling: Let $\tilde{C}^i = ((\tilde{g}^j_{a,b})^i)_{j,a,b,g} \in \{0,1\}^{4n\kappa|G|}$ be P_i's share of the whole garbled circuit.

1. Each party P_i samples random seeds $s^i_j \leftarrow \{0,1\}^\kappa$, $j \neq i$. P_i sends s^i_j to P_j over a private channel.
2. P_i computes the shares $S^i_i = \bigoplus_{i \neq j} \mathcal{G}(s^i_j)$, and $S^j_i = \mathcal{G}(s^j_i)$, for $j \neq i$.[a]
3. Each P_i, for $i = 2, \ldots, n$, sends $\tilde{C}^i \oplus \bigoplus_{j=1}^n S^j_i$ to P_1.
4. P_1 reconstructs the garbled circuit, $\tilde{C}$, and broadcasts this.

[a] Steps 1 to 2 are independent of $\tilde{C}^i$, so can be merged with previous rounds in the **Garbling** stage.

Figure 4.7: Open Garbling stage of the preprocessing protocol.

the masks for all output wires using $\mathcal{F}_{\text{Bit}\times\text{Bit}}$, so that the outputs can be obtained in the online phase.

Before opening the garbled circuit, the parties must rerandomize their shares by distributing a fresh, random secret-sharing of each share to the other parties, via private channels. This is needed so that the shares do not leak any information on the PRF values, so we can prove security. This may seem unnecessary, since the inclusion of the PRF values in the shares should randomize them sufficiently. However, we cannot prove this intuition: the same PRF values are used to compute the garbled circuit that is output by the protocol, so they cannot also be used as a one-time pad.[2] In steps 1 to 2 of Figure 4.7, we show how to perform this extra rerandomization step with $O(n^2 \cdot \kappa)$ communication.

Finally, to reconstruct the garbled circuit, the parties sum up and broadcast the rerandomized shares and add them together to get $\tilde{g}^j_{a,b}$.

4.3.3 Bit/String Multiplications

Our method for this is in the subrotocol $\Pi_{\text{Bit}\times\text{String}}$ (Figure 4.8). It proceeds in two stages: first the **Multiply** step creates the shared products, then the **Consistency Check** verifies that the correct inputs were used to create the products.

Recall that the task is for the parties to obtain an additive sharing of the products, for each $j \in [n]$ and $(a,b) \in \{0,1\}^2$:

$$\boldsymbol{r}^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \tag{4.3}$$

where the string $\boldsymbol{r}^j$ is known only to P_j, and fixed for every gate. Denote by x one of the additively shared $\lambda_{(\cdot)}$ bits used in a single bit/string product and stored by $\mathcal{F}_{\text{Bit}\times\text{Bit}}$. We obtain shares of $x \cdot \boldsymbol{r}^j$ using actively secure correlated OT (see Figure 4.3), as follows:

[2] Furthermore, the environment sees all of the PRF keys of the honest parties, since these are outputs of the protocol, which seems to rule out any kind of computational reduction in the security proof.

1. For each $i \neq j$, parties P_i and P_j run a correlated OT, with choice bit x^i and correlation $\boldsymbol{r}^j$. P_i obtains $T_{i,j}$ and P_j obtains $Q_{i,j}$ such that:

$$T_{i,j} = Q_{i,j} + x^i \cdot \boldsymbol{r}^j.$$

2. Each P_i, for $i \neq j$, defines the share $Z^i = T_{i,j}$, and P_j defines $Z^j = \sum_{i \neq j} Q_{i,j} + x^j \cdot \boldsymbol{r}^j$. Now we have:

$$\sum_{i=1}^{n} Z^i = \sum_{i \neq j} T_{i,j} + \sum_{i \neq j} Q_{i,j} + x^j \cdot \boldsymbol{r}^j \quad = \sum_{i \neq j} (T_{i,j} + Q_{i,j}) + x^j \cdot \boldsymbol{r}^j = x \cdot \boldsymbol{r}^j$$

as required.

The above method is performed $3|G|$ times and for each P_j, to produce the shared bit/string products $x \cdot \boldsymbol{r}^j$, for $x \in \{\lambda_u, \lambda_v, \lambda_{uv}\}$.

4.3.4 Consistency Check

We now show how the parties verify that the correct shares of x and correlations $\boldsymbol{r}^j$ were used in the correlated OTs, and analyse the security of this check. The parties first create $m + s$ bit/string products, where m is the number of products needed and s is a statistical security parameter, and then open random linear combinations (over $\mathbb{F}_2$) of all the products and check correctness of the opened results. This is possible because the products are just a linear function of the fixed string $\boldsymbol{r}^j$. In more detail, the parties first sample a random ε-almost 1-universal hash function $\boldsymbol{H} \leftarrow \mathbb{F}_2^{m \times s}$, and then open

$$\boldsymbol{c}_x = \boldsymbol{H} \cdot \boldsymbol{x} + \hat{\boldsymbol{x}}$$

using $\mathscr{F}_{\text{Bit}\times\text{Bit}}$. Here, $\boldsymbol{x}$ is the vector of all m wire masks to be multiplied, whilst $\hat{\boldsymbol{x}} \in \mathbb{F}_2^s$ are the additional, random masking bits, used as a one-time pad to ensure that $\boldsymbol{c}_x$ does not leak information on $\boldsymbol{x}$.

To verify that a single shared matrix $\boldsymbol{Z}_j$ is equal to $\boldsymbol{x} \otimes R^j$ (as in Figure 4.8), each party P_i, for $i \neq j$, then commits to $\boldsymbol{H} \cdot \boldsymbol{Z}_j^i$, whilst P_j commits to $\boldsymbol{H} \cdot \boldsymbol{Z}_j^j + \boldsymbol{c}_x \otimes \boldsymbol{r}^j$. The parties then open all commitments and check that these sum to zero, which should happen if the products were correct.

The intuition behind the check is that any errors present in the original bit/string products will remain when multiplied by $\boldsymbol{H}$, except with probability ε, by the almost-1-universal property (Definition 4.2). Furthermore, it turns out that cancelling out any non-zero errors in the check requires either guessing an honest party's global difference $\boldsymbol{r}^j$, or guessing the secret masking bits $\hat{\boldsymbol{x}}$. We formalize this, by first considering the exact deviations that are possible by a corrupt P_j in $\Pi_{\text{Bit}\times\text{String}}$. These are:

Bit/string multiplication subprotocol – $\Pi_{\text{Bit}\times\text{String}}$

Inputs: Each P_j inputs the private global difference string $\boldsymbol{r}^j \in \mathbb{F}_2^\kappa$, which was generated in the main protocol. All parties input $3|G|$ authenticated, additively shared bits, $\langle x_1\rangle,\ldots,\langle x_{3|G|}\rangle$, and s additional, random shared bits, $\langle \hat{x}_1\rangle,\ldots,\langle \hat{x}_s\rangle$, to be used as masking values and discarded.

I: Init: Every ordered pair of parties (P_i,P_j) calls **Initialize** on $\mathscr{F}_{\Delta\text{-ROT}}$, where P_j, the sender, inputs the global difference string R^j.

II: Multiply: For each $j \in [n]$, the parties do as follows:

1. For every $i \neq j$, parties P_i and P_j call **Extend** on the $\mathscr{F}_{\Delta\text{-ROT}}$ instance where P_j is sender, and P_i inputs the choice bits $\boldsymbol{x}^i = (x_1^i,\ldots,x_{3|G|}^i,\hat{x}_1^i,\ldots,\hat{x}_s^i)$.
   For each OT between (P_i,P_j), P_j receives $q \in \{0,1\}^\kappa$ and P_i receives $t \in \{0,1\}^\kappa$. P_i stores their $3|G|+s$ strings from this instance into the rows of a matrix $\boldsymbol{T}_{i,j}$, and P_j stores the corresponding outputs in $\boldsymbol{Q}_{i,j}$. These satisfy
   $$\boldsymbol{T}_{i,j} = \boldsymbol{Q}_{i,j} + \boldsymbol{x}^i \otimes \boldsymbol{r}^j \in \mathbb{F}_2^{(3|G|+s)\times\kappa}.$$
2. Each P_i, for $i \neq j$, defines the matrix $\boldsymbol{Z}_j^i = \boldsymbol{T}_{i,j}$, and P_j defines $\boldsymbol{Z}_j^j = \sum_{i\neq j} \boldsymbol{Q}_{i,j} + \boldsymbol{x}^j \otimes \boldsymbol{r}^j$.

Now, it should hold that $\sum_{i=1}^n \boldsymbol{Z}_j^i = \boldsymbol{x} \otimes \boldsymbol{r}^j$, for each $j \in [n]$.

III: Consistency Check: The parties check correctness of the above as follows:

1. Each P_i removes the last s rows from $\boldsymbol{Z}_j^i$ (for $j \in [n]$) and places these 'dummy' masking values in a matrix $\hat{\boldsymbol{Z}}_j^i \in \mathbb{F}_2^{s\times\kappa}$. Similarly, redefine $\boldsymbol{x}^i = (x_1^i,\ldots,x_{3|G|}^i)$ and let $\hat{\boldsymbol{x}}^i = (\hat{x_1}^i,\ldots,\hat{x_s}^i)$.
2. The parties call $\mathscr{F}_{\text{Rand}}$ (Figure 4.2) to sample a seed for a uniformly random, ε-almost 1-universal linear hash function, $\boldsymbol{H} \in \mathbb{F}_2^{s\times 3|G|}$.
3. All parties compute the vector:
   $$\langle \boldsymbol{c}_x\rangle = \boldsymbol{H}\cdot\langle \boldsymbol{x}\rangle \ + \langle \hat{\boldsymbol{x}}\rangle \in \mathbb{F}_2^s$$
   and open $\boldsymbol{c}_x$ using the **Open** command of $\mathscr{F}_{\text{Bit}\times\text{Bit}}$. If $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ aborts, the parties abort.
4. Each party P_i calls **Commit** on $\mathscr{F}_{\text{Commit}}$ (Figure 4.1) with input the n matrices:
   $$\boldsymbol{C}_j^i = \boldsymbol{H}\cdot\boldsymbol{Z}_j^i + \hat{\boldsymbol{Z}}_j^i, \quad \text{for } j\neq i, \text{ and } \quad \boldsymbol{C}_i^i = \boldsymbol{H}\cdot\boldsymbol{Z}_i^i + \hat{\boldsymbol{Z}}_i^i + \boldsymbol{c}_x \otimes \boldsymbol{r}^i.$$
5. All parties open their commitments and check that, for each $j \in [n]$
   $$\sum_{i=1}^n \boldsymbol{C}_j^i = 0.$$
   If the check fails, the parties abort.
6. Each party P_i stores the matrices $\boldsymbol{Z}_1^i,\ldots,\boldsymbol{Z}_n^i$.

Figure 4.8: Subprotocol for bit/string multiplication and checking consistency.

1. Provide inconsistent inputs $\boldsymbol{r}^j$ when acting as sender in the **Initialize** command of the $\mathscr{F}_{\Delta\text{-ROT}}$ instances with two different honest parties.

2. Input an incorrect share x^j when acting as receiver in the **Extend** command of $\mathscr{F}_{\Delta\text{-ROT}}$.

Note that in both of these cases, we are only concerned when the other party in the $\mathscr{F}_{\Delta\text{-ROT}}$ execution is honest, as if both parties are corrupt then $\mathscr{F}_{\Delta\text{-ROT}}$ does not need to be simulated in the security proof.

We model these two attacks by defining $\boldsymbol{r}^{j,i}$ and $x^{j,i}$ to be the *actual* inputs used by a corrupt P_j in the above two cases, and then define the errors (for $j \in I$ and $i \notin I$):

$$\begin{aligned}\Delta^{j,i} &= \boldsymbol{r}^{j,i} + \boldsymbol{r}^j \\ \delta_\ell^{j,i} &= x_\ell^{j,i} + x_\ell^j, \quad \ell \in [3|G|].\end{aligned}$$

Note that $\Delta^{j,i}$ is fixed in the initialization of $\mathscr{F}_{\Delta\text{-ROT}}$, whilst $\delta_\ell^{j,i}$ may be different for every OT. Whenever P_i and P_j are both corrupt, or both honest, for convenience we define $\Delta^{j,i} = 0$ and $\delta^{j,i} = 0$. This means the outputs of $\mathscr{F}_{\Delta\text{-ROT}}$ with (P_i, P_j) then satisfy (omitting ℓ subscripts)

$$t_{i,j} = q_{i,j} + x^i \cdot \boldsymbol{r}^j + \delta^{i,j} \cdot \boldsymbol{r}^j + \Delta^{j,i} \cdot x^i,$$

where $\delta^{i,j} \neq 0$ if P_i cheated, and $\Delta^{j,i} \neq 0$ if P_j cheated.

Now, as in step 1 of the first stage of $\Pi_{\text{Bit}\times\text{String}}$, we can put the $\mathscr{F}_{\Delta\text{-ROT}}$ outputs for each party into the rows of a matrix, and express the above as:

$$\boldsymbol{T}_{i,j} = \boldsymbol{Q}_{i,j} + \boldsymbol{x}^i \otimes \boldsymbol{r}^j + \boldsymbol{\delta}^{i,j} \otimes \boldsymbol{r}^j + \boldsymbol{x}^i \otimes \Delta^{j,i}$$

where $\boldsymbol{\delta}^{j,i} = (\delta_1^{j,i}, \ldots, \delta_{3|G|}^{j,i})$, and the tensor product notation is defined in Section 4.2.

Accounting for these errors in the outputs of the **Multiply** step in $\Pi_{\text{Bit}\times\text{String}}$, we get:

$$\boldsymbol{Z}_j = \sum_{i=1}^{n} \boldsymbol{Z}_j^i = \boldsymbol{x} \otimes \boldsymbol{r}^j + \underbrace{\sum_{i \in I} \boldsymbol{\delta}^{i,j}}_{=\boldsymbol{\delta}^j} \otimes \boldsymbol{r}^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}. \tag{4.4}$$

The following lemma demonstrates that if a party cheated, then to pass the check they must either guess all of the shares $\hat{\boldsymbol{x}}^i \in \mathbb{F}_2^s$ for some honest P_i, or guess P_i's global difference R^i (except with negligible probability over the choice of the ε-almost 1-universal hash function, $\boldsymbol{H}$).

Lemma 4.3.1. *If the check in* $\Pi_{\text{Bit}\times\text{String}}$ *passes, then except with probability* $\max(2^{-s}, \varepsilon + 2^{-\kappa})$*, all of the errors* $\boldsymbol{\delta}^j, \Delta^{i,j}$ *are zero.*

Proof. From (4.4), we have, for each $j \in [n]$:

$$\boldsymbol{Z}_j = \sum_{i=1}^{n} \boldsymbol{Z}_j^i = \boldsymbol{x} \otimes \boldsymbol{r}^j + \boldsymbol{\delta}^j \otimes \boldsymbol{r}^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}.$$

Notice that steps 4–5 of the check in Figure 4.8 perform n individual checks on the matrices $\boldsymbol{Z}_1,\ldots,\boldsymbol{Z}_n$ in parallel. Fix j, and first consider the check for a single matrix $\boldsymbol{Z}_j$. From here, we omit the subscript j to simplify notation.

Let $(\boldsymbol{C}^*)^i$, for $i \in I$, be the values committed to by corrupt parties in step 4, and define

$$\tilde{\boldsymbol{C}}^i = \begin{cases} \boldsymbol{H}\cdot\boldsymbol{Z}^i + \hat{\boldsymbol{Z}}^i, & \text{if } i \neq j \\ \boldsymbol{H}\cdot\boldsymbol{Z}^i + \hat{\boldsymbol{Z}}^i + \boldsymbol{c}_x \otimes \boldsymbol{r}^i, & \text{if } i = j \end{cases}$$

to be the value which a corrupt P_i *should have* committed to.

Denote the *difference* between what a corrupt P_i actually committed to, and what they should have committed to, by:

$$\boldsymbol{D}^i = (\boldsymbol{C}^*)^i + \tilde{\boldsymbol{C}}^i \in \mathbb{F}_2^{s\times\kappa}.$$

Also, define the sum of the differences $\boldsymbol{D}_I = \sum_{i\in I}\boldsymbol{D}^i$. To pass the consistency check, it must hold that $0 = \sum_{i\notin I}\boldsymbol{C}^i + \sum_{i\in I}\tilde{\boldsymbol{C}}^i + \boldsymbol{D}_I$ or, equivalently:

$$\begin{aligned} \boldsymbol{D}_I &= \sum_{i\notin I}\boldsymbol{C}^i + \sum_{i\in I}\tilde{\boldsymbol{C}}^i \\ &= \boldsymbol{H}(\sum_{i=1}^{n}\boldsymbol{Z}^i) + \sum_{i=1}^{n}\hat{\boldsymbol{Z}}^i + \boldsymbol{c}_x\otimes\boldsymbol{r}^j \\ &= \boldsymbol{H}(\boldsymbol{x}\otimes\boldsymbol{r}^j + \boldsymbol{\delta}^j\otimes\boldsymbol{r}^j + \sum_{i\notin I}\boldsymbol{x}^i\otimes\Delta^{j,i}) + \hat{\boldsymbol{Z}} + \boldsymbol{H}(\boldsymbol{x})\otimes\boldsymbol{r}^j + \hat{\boldsymbol{x}}\otimes\boldsymbol{r}^j \qquad (4.5) \\ &= \boldsymbol{H}(\boldsymbol{\delta}^j\otimes\boldsymbol{r}^j + \sum_{i\notin I}\boldsymbol{x}^i\otimes\Delta^{j,i}) + \hat{\boldsymbol{Z}} + \hat{\boldsymbol{x}}\otimes\boldsymbol{r}^j \qquad (4.6) \end{aligned}$$

where (4.5) holds because $\boldsymbol{c}_x = \boldsymbol{H}(\boldsymbol{x}) + \hat{\boldsymbol{x}}$, and (4.6) due to the linearity of $\boldsymbol{H}$ and Fact 4.2.1.

Now, taking into account the fact that $\hat{\boldsymbol{Z}}$ is constructed the same way as $\boldsymbol{Z}$, and considering (4.4), there exist some adversarially chosen errors $\hat{\boldsymbol{\delta}}^j \in \mathbb{F}_2^s$ such that:

$$\hat{\boldsymbol{Z}} = \hat{\boldsymbol{x}}\otimes\boldsymbol{r}^j + \hat{\boldsymbol{\delta}}^j\otimes\boldsymbol{r}^j + \sum_{i\notin I}\hat{\boldsymbol{x}}^i\otimes\Delta^{j,i}.$$

Plugging this into equation (4.6), the check passes if and only if:

$$\begin{aligned} \boldsymbol{D}_I &= \boldsymbol{H}(\boldsymbol{\delta}^j\otimes\boldsymbol{r}^j + \sum_{i\notin I}\boldsymbol{x}^i\otimes\Delta^{j,i}) + \hat{\boldsymbol{\delta}}^j\otimes\boldsymbol{r}^j + \sum_{i\notin I}\hat{\boldsymbol{x}}^i\otimes\Delta^{j,i} \\ &= (\boldsymbol{H}(\boldsymbol{\delta}^j) + \hat{\boldsymbol{\delta}}^j)\otimes\boldsymbol{r}^j + \sum_{i\notin I}(\boldsymbol{H}(\boldsymbol{x}^i) + \hat{\boldsymbol{x}}^i)\otimes\Delta^{j,i}. \qquad (4.7) \end{aligned}$$

We now show that the probability of this holding is negligible, if any errors are non-zero.

First consider the left-hand summation in (4.7), supposing that at least one of $\boldsymbol{\delta}^j, \hat{\boldsymbol{\delta}}^j$ is non-zero. Recall that $\boldsymbol{\delta}^j$ and $\hat{\boldsymbol{\delta}}^j$ are fixed by the adversary's inputs to $\mathscr{F}_{\Delta\text{-ROT}}$, so are independent of

the random choice of the hash function $\boldsymbol{H}$. Therefore, by the ε-almost 1-universal property of the family of linear hash functions (Definition 4.2), it holds that

$$\Pr_{\boldsymbol{H}}[\boldsymbol{H}(\boldsymbol{\delta}^j) + \hat{\boldsymbol{\delta}}^j = 0] \leq \varepsilon.$$

So except with probability ε, $\mathscr{A}$ will have to guess $\boldsymbol{r}^j$ to construct $\boldsymbol{D}_I$ to pass the check, since $\boldsymbol{r}^j$ is independent of the right-hand summation. By a union bound, therefore, if at least one of $\boldsymbol{\delta}^j$ or $\hat{\boldsymbol{\delta}}^j$ is non-zero then the check passes with probability at most $\varepsilon + 2^{-\kappa}$.

Now suppose that $\boldsymbol{\delta}^j = \hat{\boldsymbol{\delta}}^j = 0$, so $\mathscr{A}$ only needs to guess the right-hand summation of (4.7) to pass the check. If the error $\Delta^{j,i} \neq 0$ for some $i \notin I$ then the adversary must successfully guess $\boldsymbol{H}(\boldsymbol{x}^i) + \hat{\boldsymbol{x}}^i$ to be able to pass the check. Since $\hat{\boldsymbol{x}}^i$ is uniformly random in the view of the adversary, this can only occur with probability 2^{-s}.

In conclusion, the probability of passing the j-th check when any of the errors $\boldsymbol{\delta}^j, \hat{\boldsymbol{\delta}}^j$ or $\Delta^{j,i}$ are non-zero, is no more than $\max(2^{-s}, \varepsilon + 2^{-\kappa})$. Since the adversary must pass all n checks to prevent an abort, this also gives an upper bound for the overall success probability. ■

4.3.5 Security Proof

We now give some intuition behind the security of the whole protocol. In the proof, the strategy of the simulator is to run an internal copy of the protocol, using dummy, random values for the honest parties' keys and wire mask shares. All communication with the adversary is simulated by computing the correct messages according to the protocol and the dummy honest shares, until the final output stage. In the output stage, we switch to fresh, random honest parties' shares, consistent with the garbled circuit received from $\mathscr{F}_{\text{Preprocessing}}$ and the corrupt parties' shares.

Firstly, by Lemma 4.3.1, it holds that in the real execution, if the adversary introduced any non-zero errors then the consistency check fails with overwhelming probability. The same is true in the ideal execution; note that the errors are still well-defined in this case because the simulator can compute them by comparing all inputs received to $\mathscr{F}_{\Delta\text{-ROT}}$ with the inputs the adversary should have used, based on its random tape. This implies that the probability of passing the check is the same in both worlds. Also, if the check fails then both executions abort, and it is straightforward to see that the two views are indistinguishable because no outputs are sent to honest parties (hence, also the environment).

It remains to show that the two views are indistinguishable when the consistency check passes, and the environment sees the outputs of all honest parties, as well as the view of the adversary during the protocol. The main point of interest here is the output stage. We observe that, without the final rerandomization step, the honest parties' shares of the garbled circuit would *not be uniformly random*. Specifically, consider an honest P_i's share, $(\tilde{g}^j_{a,b})^i$, where P_j is corrupt. This is computed by adding some PRF value, v, to the $\mathscr{F}_{\Delta\text{-ROT}}$ outputs where P_i was receiver and P_j was sender (step 2 of $\Pi_{\text{Bit}\times\text{String}}$). Since P_j knows both strings in each OT, there are only two possibilities for P_i's output (depending on the choice bit), so this is not uniformly

random. It might be tempting to argue that v is a random PRF output, so serves as a one-time pad, but this proof attempt fails because v is also used to compute the final garbled circuit. In fact, it seems difficult to rely on any reduction to the PRF, since all the PRF keys are included in the output to the environment.

To avoid this issue, we need the rerandomization step using a PRG, and the additional assumption of secure point-to-point channels. Note that this is missing from the protocol of [25], which does not currently have a security proof. Rerandomization ensures that the honest shares can be simulated with random values which, together with the corrupt shares, sum up to the correct garbled circuit. We proceed with the complete proof.

Theorem 4.3.1. *Protocol* $\Pi_{\text{Preprocessing}}$ *from Figure 4.6 UC-securely computes* $\mathcal{F}_{\text{Preprocessing}}$ *from Figure 4.5 in the presence of a static, active adversary corrupting up to* $n-1$ *parties in the* $\{\mathcal{F}_{\Delta\text{-ROT}}, \mathcal{F}_{\text{Bit}\times\text{Bit}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Commit}}\}$*-hybrid model.*

Proof. Let $\mathcal{A}$ denote a PPT adversary corrupting a strict subset I of parties. As part of the proof, we will construct a simulator $\mathcal{S}$ that plays the roles of the honest parties on arbitrary inputs, as well as the roles of functionalities $\{\mathcal{F}_{\Delta\text{-ROT}}, \mathcal{F}_{\text{Bit}\times\text{Bit}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Commit}}\}$ and interacts with $\mathcal{A}$. We assume w.l.o.g. that $\mathcal{A}$ is a deterministic adversary, which receives as additional input a random tape that determines its internal coin tosses. Nevertheless, since $\mathcal{A}$ is active, it may still ignore the random tape, and use its own (possibly biased) random values instead.

The simulator begins by initializing $\mathcal{A}$ with the inputs from the environment, $\mathcal{Z}$, and a uniform string r as its random tape. During the simulation, we will use r to compute values that $\mathcal{A}$ *should* (but might not) use during the protocol. We can now define the rest of $\mathcal{S}$ as follows:

Garbling:

1. The simulator emulates $\mathcal{F}_{\Delta\text{-ROT}}$.`Initialize` as follows:
   - For a honest party $P_i, i \notin I$, $\mathcal{S}$ samples $\boldsymbol{r}^i \in \{0,1\}^\kappa$.
   - For a corrupted party $P_j, j \in I$, $\mathcal{S}$ computes $\boldsymbol{r}^j$ from that party's random tape. Additionally, for each pair of parties involving that party, (P_j, P_i) where $i \notin I$, $\mathcal{S}$ receives by $\mathcal{A}$ values $\boldsymbol{r}^{j,i} \in \{0,1\}^\kappa$ and stores $\Delta^{j,i} = \boldsymbol{r}^{j,i} + \boldsymbol{r}^j$. If $\forall i \notin I, \Delta^{j,i} = d$, then $\mathcal{S}$ modifies its stored values by setting $\boldsymbol{r}^j \leftarrow \boldsymbol{r}^j + d$ and $\Delta^{j,i} \leftarrow 0, \forall i \notin I$.
2. GENERATE WIRE MASKS AND KEYS: Passing through each wire w of the circuit topologically, the simulator $\mathcal{S}$ proceeds as follows.
   - It emulates $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ and defines the wire masks:
     - If w is a *circuit-input* wire, P_i is the party whose input is associated with it and $i \in I$, then $\mathcal{S}$ receives λ_w by $\mathcal{A}$. If $i \notin I$, then $\mathcal{S}$ chooses λ_w.
     - If w is the output of an AND gate, $\mathcal{S}$ samples $\lambda_w \leftarrow \mathbb{F}_2$ and receives from the adversary λ_w^i for every $i \in I$. Then, $\mathcal{S}$ samples random λ_w^j, $j \notin I$, such that $\lambda_w = \sum_{\ell \in [n]} \lambda_w^\ell$.

  - If w is the output of a XOR gate, $\mathscr{S}$, it sets $\lambda_w = \lambda_u + \lambda_v$.
- It defines the PRF keys:
  - If w is not the output of a XOR gate, $\mathscr{S}$ computes $k^i_{w,0} \in \{0,1\}^\kappa$ for $i \in [n]$. For corrupted parties $i \in I$, $\mathscr{S}$ reads this off P_i's random tape, whereas for honest parties it samples a random key.
  - If w is the output of a XOR gate, for $i \in [n]$, it sets $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i$ for $i \in [n]$.
- Finally, it sends to $\mathscr{F}_{\text{Preprocessing}}$ the global difference R^i and the keys $k^i_{w,0}$, for each $i \in I$ and each w that is an output wire of an AND gate, as well as the wire masks λ_w (for each w that is an input wire of a corrupt party).

3. SECURE PRODUCT COMPUTATIONS: The simulator $\mathscr{S}$ emulates $\mathscr{F}_{\text{Bit}\times\text{Bit}}$.`Input` receiving $\hat{\boldsymbol{x}}^i = (\hat{x}^i_1, \ldots, \hat{x}^i_s)$ for every $i \in I$, and also samples honest parties' shares $\hat{\boldsymbol{x}}^j$, for $j \notin I$. For each AND gate $g \in G$, it emulates $\mathscr{F}_{\text{Bit}\times\text{Bit}}$.`Multiply` by receiving shares $(\lambda_{uv})^i$ from $\mathscr{A}$ for $i \in I$ and setting random $(\lambda_{uv})^j$ for $j \notin I$ such that $\sum_{\ell\in[n]}(\lambda_{uv})^\ell = \lambda_u \cdot \lambda_v$. For the $\Pi_{\text{Bit}\times\text{String}}$ subprotocol:

   - Multiply: $\mathscr{S}$ emulates $\mathscr{F}_{\Delta\text{-ROT}}$.`Extend` between each pair of parties P_i and P_j. If both parties are honest, or if both are corrupted, the simulation is trivial. Hereafter, we focus on the cases where exactly one party of each pair is corrupted:
     - When P_i is a corrupted sender, $\mathscr{S}$ receives a (possibly) different $q_i \in \{0,1\}^\kappa$ from $\mathscr{A}$ for each of the $3|G| + s$ calls.
     - When P_i is a corrupted receiver, $\mathscr{S}$ receives values $\hat{x}^i_1 + \delta^{i,j}_{\hat{x}_1}, \ldots, \hat{x}^i_s + \delta^{i,j}_{\hat{x}_s}$ and, for each AND gate, $\left(\lambda^i_u + \delta^{i,j}_{\lambda_u}, \lambda^i_v + \delta^{i,j}_{\lambda_v}, \lambda^i_{uv} + \lambda^i_w + \delta^{i,j}_{\lambda_{uv}+\lambda_w}\right)_{(u,v,w)}$. For each of the $3|G| + s$ previous inputs, it also receives a (possibly) different $t_i \in \{0,1\}^\kappa$ from $\mathscr{A}$.

     Note that the errors $\delta^{i,j}$ and the t_i, q_i values received from $\mathscr{A}$ are stored by $\mathscr{S}$, whilst the λ values and shares were fixed in the previous stage.
   - Consistency Check:

     2. $\mathscr{S}$ emulates $\mathscr{F}_{\text{Rand}}$ and sends a seed for a uniformly random ε-almost 1-universal linear hash function $\boldsymbol{H} \in \mathbb{F}_2^{s\times 3|G|}$ to $\mathscr{A}$.
     3. $\mathscr{S}$ emulates $\mathscr{F}_{\text{Bit}\times\text{Bit}}$.`Open` and sends $\boldsymbol{c}_x = \boldsymbol{H} \cdot \boldsymbol{x} + \hat{\boldsymbol{x}} \in \mathbb{F}_2^s$ to $\mathscr{A}$, where $\boldsymbol{x}$ is the vector of $3|G|$ values $(\lambda_u, \lambda_v, \lambda_{uvw})_{(u,v,w)}$ from the previous stage, and $\hat{\boldsymbol{x}} = \sum_{i\in[n]} \hat{\boldsymbol{x}}^i$, received just before the **Multiply** step. If $\mathscr{A}$ does not send back `OK` to $\mathscr{S}$, then $\mathscr{S}$ sends $\perp$ to $\mathscr{F}_{\text{Preprocessing}}$ and aborts.

     4-5. Emulating $\mathscr{F}_{\text{Commit}}$, $\mathscr{S}$ receives $\boldsymbol{C}^i_\ell$ from $\mathscr{A}$ for all $\ell \in [n]$ and $i \in I$. It then computes $\boldsymbol{C}^j_\ell$ for $j \notin I$, as each honest party would, and completes the emulation of $\mathscr{F}_{\text{Commit}}$ by sending these to $\mathscr{A}$. If $\sum_{i\in I} \boldsymbol{C}^i_\ell + \sum_{j\notin I} \boldsymbol{C}^j_\ell \neq 0$, for any $\ell \in [n]$, $\mathscr{S}$ sends $\perp$ to $\mathscr{F}_{\text{Preprocessing}}$ and terminates.

4. GARBLE GATES: For every AND gate $g \in G$, the simulator $\mathscr{S}$ computes and stores corrupt parties' shares of the garbled circuit, $\tilde{C}^i$, for $i \in I$, as the adversary should do according to the protocol. Note that $\mathscr{S}$ has all the necessary values to do so from the messages it previously received and the knowledge of $\mathscr{A}$'s random tape. Namely, the simulator knows the PRF keys, the global differences, and the t_i and q_i values received in the $\mathscr{F}_{\Delta\text{-ROT}}$ calls.
5. REVEAL MASKS FOR OUTPUT WIRES: $\mathscr{S}$ emulates $\mathscr{F}_{\text{Bit}\times\text{Bit}}$.Open and for every circuit-output wire w, it calls $\mathscr{F}_{\text{Preprocessing}}$ to get the wire mask λ_w and forward it to $\mathscr{A}$. If $\mathscr{A}$ does not send back OK to $\mathscr{S}$, then $\mathscr{S}$ sends $\perp$ to $\mathscr{F}_{\text{Preprocessing}}$ and terminates.

Open Garbling:

6. For each $i \in I$, $\mathscr{S}$ samples random shares $\{S^i_j\}_{j \notin I}$ and sends these to $\mathscr{A}$.
7. $\mathscr{S}$ then calls $\mathscr{F}_{\text{Preprocessing}}$ to receive the garbled circuit $\tilde{C}$. Using the corrupted parties' shares $\tilde{C}^i$, $i \in I$, received previously, $\mathscr{S}$ generates random honest parties' shares $\tilde{C}^j$, $j \notin I$, subject to the constraint that $\sum_{\ell \in [n]} \tilde{C}^\ell = \tilde{C}$. Once this is done, $\mathscr{S}$ forwards the honest shares of the garbled circuit to $\mathscr{A}$. If $\mathscr{A}$ does not respond with OK then $\mathscr{S}$ sends $\perp$ to $\mathscr{F}_{\text{Preprocessing}}$ and terminates. Otherwise, it receives from the adversary OK and shares $\hat{C}^i$ for $i \in I$. Finally, $\mathscr{S}$ computes the error $E = \sum_{i \in I}(\tilde{C}^i + \hat{C}^i)$ and sends this to $\mathscr{F}_{\text{Preprocessing}}$.

INDISTINGUISHABILITY: We will first show that, during the **Garbling** phase, the environment $\mathscr{Z}$ cannot distinguish between an interaction with $\mathscr{S}$ and $\mathscr{F}_{\text{Preprocessing}}$ and an interaction with the real adversary $\mathscr{A}$ and $\Pi_{\text{Preprocessing}}$. We then argue that the garbled circuit, and the honest parties' shares of it, are also identically distributed in both worlds.

Garbling phase indistinguishability: Let's look at the **Garbling** command. In both worlds and for every AND gate, the honest parties's shares for the masks λ_w, and for the products λ_{uv} are uniformly random additive shares, whereas the corrupted parties shares' are chosen by $\mathscr{A}$. Every other step up to the execution of the $\Pi_{\text{Bit}\times\text{String}}$ subprotocol provides no output to the parties, and hence $\mathscr{Z}$ has exactly the same view in both worlds up to that point.

In the **Multiply** step of $\Pi_{\text{Bit}\times\text{String}}$, $\mathscr{S}$ only receives values from $\mathscr{A}$, so no further information is added to his view here. Note that since the corrupted parties' inputs to $\mathscr{F}_{\Delta\text{-ROT}}$ are received by $\mathscr{S}$, all the errors $\Delta^{i,j}, \boldsymbol{\delta}^j = \sum_{i \in I} \boldsymbol{\delta}^{i,j}$ are well-defined in the simulation.

Next, consider the **Consistency Check** step. If any of the errors are non-zero, then from Lemma 4.3.1, we know that in both worlds the check fails with overwhelming probability. In this case, no outputs are sent to the honest parties, and (recalling that there are no inputs from honest parties) indistinguishability is trivial since the simulator just behaved as honest parties would until this point.

We now assume that all of the errors $\Delta^{i,j}, \boldsymbol{\delta}^j = \sum_{i \in I} \boldsymbol{\delta}^{i,j}$ are zero, and so the check passes. The values $\boldsymbol{H}$ and $\boldsymbol{c}_x = \boldsymbol{H}\boldsymbol{x} + \hat{\boldsymbol{x}}$ seen by $\mathscr{A}$ are uniformly random in both worlds, since the

masking values $\hat{\boldsymbol{x}}$ are uniformly random and never seen by the environment. The distribution of the committed and opened values, $\boldsymbol{C}_j^i$, is more subtle, however. First, consider the case when there is exactly one honest party. In this case, in both worlds the values $\boldsymbol{C}_i^j$, for honest P_i, are a deterministic function of the adversary's behaviour, since they should satisfy $\sum_{i=1}^n \boldsymbol{C}_j^i = 0$. Therefore, these values are identically distributed in both worlds.

Now, suppose there is more than one honest party. The values $\boldsymbol{C}_i^i$ are computed based on the $\boldsymbol{Z}_i^i$ values, which are the sum of outputs from $\mathcal{F}_{\Delta\text{-ROT}}$ with every other party. This means for every honest P_i, $\boldsymbol{C}_i^i$ is uniformly random, since it includes an $\mathcal{F}_{\Delta\text{-ROT}}$ output with one other honest party. On the other hand, the values $\boldsymbol{C}_j^i$, where $j \in I$ and $i \notin I$, only come from a single $\mathcal{F}_{\Delta\text{-ROT}}$ instance between P_i and P_j, and P_i's output from $\mathcal{F}_{\Delta\text{-ROT}}$ in this case is not random in the view of $\mathcal{A}$. It actually should satisfy:

$$\boldsymbol{C}_j^i = \boldsymbol{H} \cdot \boldsymbol{Z}_j^i + \hat{\boldsymbol{Z}}_j^i = \boldsymbol{H} \cdot \boldsymbol{Q}_{i,j} + \hat{\boldsymbol{Q}}_{i,j} + (\boldsymbol{H} \cdot \boldsymbol{x}^i + \hat{\boldsymbol{x}}^i) \otimes \boldsymbol{r}^j$$

where $\boldsymbol{Q}_{i,j}, \hat{\boldsymbol{Q}}_{i,j}$ are the $\mathcal{F}_{\Delta\text{-ROT}}$ outputs of corrupt P_j, and $\boldsymbol{r}^j$ is also known to P_j. This means for each row of $\boldsymbol{C}_j^i$, in the view of $\mathcal{A}$ there are only two possibilities, depending on one bit from $(\boldsymbol{H} \cdot \boldsymbol{x}^i + \hat{\boldsymbol{x}}^i)$. Since the shares $\hat{\boldsymbol{x}}^i$ are uniformly random and never seen by the environment, $\boldsymbol{H} \cdot \boldsymbol{x}^i + \hat{\boldsymbol{x}}^i$ is also uniformly random in both worlds, subject to the constraint that these sum to $\boldsymbol{c}_x$ (which was opened previously). We conclude that the $\boldsymbol{C}_i^j$ values are identically distributed.

After $\Pi_{\text{Bit}\times\text{String}}$, $\mathcal{Z}$ remains unable to distinguish in the garbling phase. First, the **Garble Gates** step of $\Pi_{\text{Preprocessing}}$ requires no communication. Finally, regarding **Reveal masks for output wires**, the revealed wire masks are random bits in both worlds.

Open Garbling phase indistinguishability: The seeds sent in the first step of the **Open Garbling** stage are uniformly random in both executions. We claim that the rerandomized shares of the garbled circuit in the real execution are computationally indistinguishable from the simulated random shares. This holds because, (1) The simulated shares seen by the adversary are uniformly random, subject to the constraint that all shares sum up to the same garbled circuit, and (2) In the real world, every pair of honest parties masks their shares with outputs from the PRG $\mathcal{G}$, using a unique seed that is not seen by the environment. By a standard hybrid argument, we can therefore reduce indistinguishability of the shares to security of the PRG, by successively replacing each PRG output sent between two honest parties with a random string. ■

4.4 More Efficient Garbling with Multi-Party TinyOT

We now describe a less general, but concretely more efficient, variant of the protocol in the previous section. We replace the generic $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ functionality with a more specialized one based on 'TinyOT'-style information-theoretic MACs. This is asymptotically worse, but more practical, than using [75] for $\mathcal{F}_{\text{Bit}\times\text{Bit}}$. It also allows us to completely remove the bit/string multiplications and consistency checks in $\Pi_{\text{Bit}\times\text{String}}$, since we show that these can be obtained directly from the TinyOT MACs. This means the only cost in the protocol, apart from opening and evaluating the

garbled circuit, is the single bit multiplication per AND gate in the underlying TinyOT-based protocol.

For completeness, we present a complete description of a suitable TinyOT-based protocol in Section 4.7. This is done by combining the multiplication triple generation protocol (over $\mathbb{F}_2$) from [58] with a consistency check to enforce correct shared random bits, which is similar to the more general check from the previous section.

4.4.1 Secret-Shared MAC Representation

For $x \in \{0,1\}$ held by P_i, define the following two-party MAC representation, as used in 2-party TinyOT [104]:

$$[x]_{i,j} = (x, M_j^i, K_i^j), \qquad M_j^i = K_i^j + x \cdot R^j$$

where P_i holds x and a MAC M_j^i, and P_j holds a local MAC key K_i^j, as well as the fixed, global MAC key R^j. Similarly, we define the n-party representation of an additively shared value $x = x^1 + \cdots + x^n$:

$$[x] = (x^i, \{M_j^i, K_j^i\}_{j \neq i})_{i \in [n]}, \quad M_j^i = K_i^j + x^i \cdot R^j$$

where each party P_i holds the $n-1$ MACs M_j^i on x^i, as well as the keys K_j^i on each x^j, for $j \neq i$, and a global key R^i. Note that this is equivalent to every pair (P_i, P_j) holding a representation $[x^i]_{i,j}$.

The key observation for this section, is that a sharing $[x]$ can be used to directly compute shares of all the products $x \cdot R^j$, as in the following claim.

Claim 4.4.1. *Given a representation $[x]$, the parties can locally compute additive shares of $x \cdot R^j$, for each $j \in [n]$.*

Proof. Write $[x] = (x^i, \{M_j^i, K_j^i\}_{j \neq i})_{i \in [n]}$. Each party P_i defines the n shares

$$Z_i^i = x^i \cdot R^i + \sum_{j \neq i} K_j^i \quad \text{and} \quad Z_j^i = M_j^i, \quad \text{for each } j \neq i.$$

We then have, for each $j \in [n]$:

$$\sum_{i=1}^{n} Z_j^i = Z_j^j + \sum_{i \neq j} Z_j^i = (x^j \cdot R^j + \sum_{i \neq j} K_i^j) + \sum_{i \neq j} M_j^i =$$
$$= x^j \cdot R^j + \sum_{i \neq j} (M_j^i + K_i^j) = x^j \cdot R^j + \sum_{i \neq j} (x^i \cdot R^j) = x \cdot R^j.$$

■

We define addition of two shared values $[x], [y]$, to be straightforward addition of the components. We define addition of $[x]$ with a public constant $c \in \mathbb{F}_2$ by:

Functionality $\mathcal{F}_{\text{n-TinyOT}}$

Initialize: On receiving (`init`) from all parties, the functionality receives $R^i \in \{0,1\}^\kappa$, for $i \in I$, from the adversary, and then samples $R^i \leftarrow \{0,1\}^\kappa$, for $i \notin I$, and sends R^i to party P_i.

Prep: On receiving $(\texttt{Prep}, m, M)$ from all parties, generate m random bits as follows:

1. Receive corrupted parties' shares $b_\ell^i \in \mathbb{F}_2$ from $\mathcal{A}$, for $i \in I$.
2. Sample honest parties' shares, $b_\ell^i \leftarrow \mathbb{F}_2$, for $i \notin A$ and $\ell \in [m]$.
3. Run n-Bracket$(b_\ell^1, \ldots, b_\ell^n)$, for every $\ell \in [m]$, so each party obtains the shares b_ℓ^i, as well as $n-1$ MACs on b_ℓ^i and a key on each b_ℓ^j, for $j \neq i$.

And M multiplication triples as follows:

1. Sample $a_\ell, b_\ell \leftarrow \mathbb{F}_2$ and compute $c_\ell = a_\ell \cdot b_\ell$, for $\ell \in [M]$.
2. For each $x \in \{a_\ell, b_\ell, c_\ell\}_{\ell \in [M]}$, authenticate x as follows:
   a) Receive corrupted parties' shares $x^i \in \mathbb{F}_2$, for $i \in I$, from $\mathcal{A}$.
   b) Sample honest parties' shares $x^i \leftarrow \mathbb{F}_2$, for $i \notin I$ subject to $\sum_{i=1}^n x^i = x$.
   c) Run n-Bracket$(x^1, \ldots, x^n)$, so the parties obtain $[x]$.

Key queries: On receiving (i, R') from $\mathcal{A}$, where $i \in [n]$, output 1 to $\mathcal{A}$ if $R^i = R'$. Otherwise, output 0 to $\mathcal{A}$.

Figure 4.9: Functionality for secure multi-party computation based on TinyOT

- P_1 stores: $(x^1 + c, \{M_j^1, K_j^1\}_{j \neq 1})$
- P_i stores: $(x^i, (M_1^i, K_1^i + c \cdot R^i), \{M_j^i, K_j^i\}_{j \in [n] \setminus \{1,i\}})$, for $i \neq 1$

This results in a correct sharing of $[x + c]$.

We can create a sharing of the product of two shared values using a random multiplication triple $([x], [y], [z])$ such that $z = x \cdot y$ with Beaver's technique [16], shown in Figure 4.18.

4.4.2 MAC-Based MPC Functionality

The functionality $\mathcal{F}_{\text{n-TinyOT}}$, which we use in place of $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ for the optimized preprocessing, is shown in Figure 4.9. It produces authenticated sharings of random bits and multiplication triples. For both of these, $\mathcal{F}_{\text{n-TinyOT}}$ first receives corrupted parties' shares, MAC values and keys from the adversary, and then randomly samples consistent sharings and MACs for the honest parties.

Another important aspect of the functionality is the **Key Queries** command, which allows the adversary to try to guess the MAC key R^i of any party, and will be informed if the guess is correct. This is needed to allow the security proof to go through; we explain this in more detail in Section 4.7. In that section we also present a complete description of a variant on the multi-party TinyOT protocol, which can be used to implement this functionality.

Macro n-Bracket

This subroutine of $\mathscr{F}_{\text{n-TinyOT}}$ uses the global MAC keys $R^1,\ldots,R^n$ stored by the functionality.

On input $(x^1,\ldots,x^n)$, authenticate the share $x^i \in \{0,1\}$, for each $i \in [n]$, as follows:

If P_i is corrupt: receive a MAC $M_j^i \in \mathbb{F}_2^\kappa$ from $\mathscr{A}$ and compute the key $K_i^j = M_j^i + x^i \cdot R^j$, for each $j \neq i$.

Otherwise:

1. Sample honest parties' keys $K_i^j \leftarrow \mathbb{F}_2^\kappa$, for $j \in [n] \setminus (I \cup \{i\})$.
2. Receive keys $K_i^j \in \mathbb{F}_2^\kappa$, for each $j \in I$, from $\mathscr{A}$.
3. Compute the MACs $M_j^i = K_i^j + x^i \cdot R^j$, for $j \in I$.

Finally, output $(x^i, \{M_j^i, K_j^i\}_{j \neq i})$ to party P_i, for $i \in [n]$.

Figure 4.10: Macro used by $\mathscr{F}_{\text{n-TinyOT}}$ to authenticate bits

4.4.3 Garbling with $\mathscr{F}_{\text{n-TinyOT}}$

Following from the observation in Claim 4.4.1, if each party P_j chooses the global difference string in $\Pi_{\text{Preprocessing}}$ to be the same R^j as in the MAC representation, then given $[\lambda]$, additive shares of the products $\lambda \cdot R^j$ can be obtained at no extra cost. Moreover, the shares are guaranteed to be correct, and the honest party's shares will be random (subject to the constraint that they sum to the correct value), since they come directly from the $\mathscr{F}_{\text{n-TinyOT}}$ functionality. This means there is no need to perform the consistency check, which greatly simplifies the protocol.

The rest of the protocol is mostly the same as $\Pi_{\text{Preprocessing}}$ in Figure 4.6, using $\mathscr{F}_{\text{n-TinyOT}}$ with $[\cdot]$-sharings instead of $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ with $\langle\cdot\rangle$-sharings. One other small difference is that because $\mathscr{F}_{\text{n-TinyOT}}$ does not have a private input command, we instead sample $[\lambda_w]$ shares for input wires using random bits, and later use a private output protocol to open the relevant input wire masks to P_i. This change is not strictly necessary, but simplifies the protocol for implementing $\mathscr{F}_{\text{n-TinyOT}}$ – if $\mathscr{F}_{\text{n-TinyOT}}$ also had an **Input** command for sharing private inputs based on n-Bracket, it would be much more complex to implement with the correct distribution of shares and MACs.

In more detail, the **Garbling** phase proceeds as follows.

1. Each party obtains a random key offset $\boldsymbol{r}^i$ by calling the **Initialize** command of $\mathscr{F}_{\text{n-TinyOT}}$.
2. For every wire w which is an input wire, or the output wire of an AND gate, the parties obtain a shared mask $[\lambda_w]$ using the **Bit** command of $\mathscr{F}_{\text{n-TinyOT}}$.
3. All the wire keys $k_{w,0}^i, k_{w,1}^i = k_{w,0}^i \oplus R^i$ are defined by P_i the same way as in $\Pi_{\text{Preprocessing}}$.
4. For XOR gates, the output wire mask is computed as $[\lambda_w] = [\lambda_u] + [\lambda_v]$.
5. For each AND gate, the parties compute $[\lambda_{uv}] = [\lambda_u \cdot \lambda_v]$ using the subprotocol Π_{Mult} in Figure 4.18.

6. The parties then obtain shares of the garbled circuit as follows:

   - For each AND gate $g \in G$ with wires (u, v, w), the parties use Claim 4.4.1 with the shared values $[\lambda_u], [\lambda_v], [\lambda_{uv} + \lambda_w]$, to define, for each $j \in [n]$, shares of the bit/string products:
   $$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad (\lambda_{uv} + \lambda_w) \cdot R^j$$
   - These are then used to define shares of $\rho_{j,a,b}$ and the garbled circuit, as in the original protocol.

7. For every circuit-output-wire w, the parties run Π_{Open} to reveal λ_w to all the parties.

8. For every *circuit input wire* w corresponding to party P_i's input, the parties run Π^i_{Open} (Figure 4.15) to open λ_w to P_i.

The only interaction introduced in the new protocol is in the multiply and opening protocols, which were abstracted away by $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ in the previous protocol. Simulating and proving security of these techniques is straightforward, due to the correctness and randomness of the multiplication triples and MACs produced by $\mathcal{F}_{\text{n-TinyOT}}$. One important detail is the **Key Queries** command of the $\mathcal{F}_{\text{n-TinyOT}}$ functionality, which allows the adversary to try to guess an honest party's global MAC key share, R^i, and learn if the guess is correct. To allow the proof to go through, we modify $\mathcal{F}_{\text{Preprocessing}}$ to also have a **Key Queries** command, so that the simulator can use this to respond to any key queries from the adversary. We denote this modified functionality by $\mathcal{F}^{\text{KQ}}_{\text{Preprocessing}}$.

The following theorem can be proven, similarly to the proof of Theorem 4.3.1 where we modify the preprocessing functionality to support key queries, and adjust the simulation as described above.

Theorem 4.4.1. *The modified protocol described above UC-securely computes $\mathcal{F}^{\text{KQ}}_{\text{Preprocessing}}$ from Figure 4.5 in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\text{n-TinyOT}}$-hybrid model.*

Finally, in Section 4.5.1 we discuss how to extend the proof of the online phase, showing that allowing key queries in the preprocessing functionality does not affect security.

4.5 The Online Phase

Our final protocol, presented in Figure 4.11, implements the online phase where the parties reveal the garbled circuit's shares and evaluate it. Our protocol is presented in the $\mathcal{F}_{\text{Preprocessing}}$-hybrid model. Upon reconstructing the garbled circuit and obtaining all input keys, the process of evaluation follows what was described in Section 2.6.1. Decrypting the 'double encrypted' garbled rows here requires each party to compute compute n^2 PRF values per gate. We recall that during

The MPC Protocol - Π_{BMR}

On input a circuit C_f representing the function f and $\rho = (\rho_1, \ldots, \rho_n)$ where ρ_i is party P_i's input, the parties execute the following commands in sequence.

Preprocessing: This sub-task is performed as follows.

- Call **Garbling** on $\mathcal{F}_{\text{Preprocessing}}$ with input C_f.
- Each party P_i obtains the λ_w wire masks for every output wire and every wire associated with their input, and all the keys $\{k^i_{w,0}\}_{w \in W}$ and $\boldsymbol{r}^i$.

Online Computation: This sub-task is performed as follows.

- For all input wires w with input from P_i, party P_i computes $\Lambda_w = \rho_w \oplus \lambda_w$, where ρ_w is P_i's input to C_f, and λ_w was obtained in the preprocessing stage. Then, P_i broadcasts the public value Λ_w to all parties.
- For all input wires w, each party P_i broadcasts the key k^i_w associated to Λ_w.
- The parties call **Open Garbling** on $\mathcal{F}_{\text{Preprocessing}}$ to reconstruct $\tilde{g}^j_{a,b}$ for every gate g and values a, b.
- Passing through the circuit topologically, the parties can now locally compute the following operations for each gate g. Let the gates input wires be labelled u and v, and the output wire be labelled w. Let a and b be the respective external values on the input wires.
  1. If g is a XOR gate, set the public value on the output wire to be $c = a \oplus b$. In addition, for every $j \in [n]$, each party computes $k^j_{w,c} = k^j_{u,a} \oplus k^j_{v,b}$.
  2. If g is an AND gate , then each party computes, for all $j \in [n]$:
  $$k^j_{w,c} = \tilde{g}^j_{a,b} \oplus \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a}, k^i_{v,b}}(g \| j) \right)$$
  3. If $k^i_{w,c} \notin \{k^i_{w,0}, k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i\}$, then P_i outputs `abort`. Otherwise, it proceeds. If P_i aborts it notifies all other parties with that information. If P_i is notified that another party has aborted it aborts as well.
  4. If $k^i_{w,c} = k^i_{w,0}$ then P_i sets $c = 0$; if $k^i_{w,c} = k^i_{w,1}$ then P_i sets $c = 1$.
  5. The output of the gate is defined to be $(k^1_{w,c}, \ldots, k^n_{w,c})$ and the public value c.
- Assuming no party aborts, everyone will obtain a public value c_w for every circuit-output wire w. The party can then recover the actual output value from $\rho_w = c_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

Figure 4.11: The MPC Protocol - Π_{BMR}.

the evaluation parties only see the randomly masked wire values and cannot determine the actual wire values. Upon completion, the parties compute the actual output using the output wire masks revealed from $\mathcal{F}_{\text{Preprocessing}}$. We conclude with the following theorem.

Theorem 4.5.1. *Let f be an n-party functionality $\{0,1\}^{n\kappa} \mapsto \{0,1\}^{\kappa}$ and assume that F is a circular 2-correlation robust PRF. Then Protocol Π_{BMR}, from Figure 4.11, UC-securely computes f in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\text{Preprocessing}}$-hybrid.*

Our proof follows by first demonstrating that the adversary's view is computationally indis-

tinguishable in both real and simulated executions. To be concrete, we consider an event for which the adversary successfully causes the bit transferred through some wire to be flipped and prove that this event can only occur with negligible probability (our proof is different to the proof in [93] as in our case the adversary may choose its additive error as a function of the garbled circuit). Then, conditioned on the event flip not occurring, we prove that the two executions are computationally indistinguishable via a reduction to the correlation robust PRF, inducing a garbled circuit that is indistinguishable. The complete proof is found below.

Proof. Let $\mathscr{A}$ be a PPT adversary corrupting a subset of parties $I \subset [n]$. We prove that there exists a PPT simulator $\mathscr{S}$ with access to an ideal functionality $\mathscr{F}$ that implements f, that simulates the adversary's view. Denoting the set of honest parties by $\bar{I}$, our simulator $\mathscr{S}$ is defined below.

DESCRIPTION OF THE SIMULATION.

1. INITIALIZATION. Upon receiving the adversary's input $(1^\kappa, I, \vec{x}_I)$, $\mathscr{S}$ incorporates $\mathscr{A}$ and internally emulates an execution of the honest parties running Π_{BMR} with the adversary $\mathscr{A}$.

2. PROCESSING. $\mathscr{S}$ emulates the preprocessing phase of functionality $\mathscr{F}_{\text{Preprocessing}}$, obtaining the adversary input $(\texttt{init}, C_f)$ where C_f is a Boolean circuit that computes f with a set of wires W and a set G of AND gates.

3. GARBLING. Let W' denote the set of input wires that are associated with the adversary's input. Then upon receiving the input $(\texttt{Garbling}, C_f)$ from the adversary the simulator emulates the garbling phase as follows.

   - Upon receiving the global differences $\{\boldsymbol{r}^i\}_{i\in I}$ from the adversary the simulator records this set.
   - For every input wire $w \in W'$ that is associated to the adversary's input, the simulator obtains from the adversary a random masking value $\lambda_w \in \{0,1\}$ and an input key $k^i_{w,0} \in \{0,1\}^\kappa$.
   - For every wire $w \in W$ that is the output of an AND gate and $i \in I$, the simulator chooses a random $\lambda_w \in \{0,1\}$ and records it. Moreover, the simulator obtains from the adversary a key $k^i_{w,0}$ and records the pair $(k^i_{w,0}, k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i)$.
   - Upon receiving an $\texttt{OK}$ command from the adversary, the simulator forwards it a random masking value $\lambda_w \in \{0,1\}$, for every output wire $w \in W$.

4. ONLINE COMPUTATION. In the online computation the simulator honestly generates the external values $\{\Lambda_w\}_{w\in W''}$ and the input keys $\{k^i_{w,\Lambda_w}\}_{i\in\bar{I}, w\in W''}$ that are associated with the honest parties' input wire set W'', and broadcasts these to the adversary.

It then obtains the adversary's external values $\{\Lambda_w\}_{w\in W'}$ as well as its input keys $\{\hat{k}^i_w\}_{i\in I,w\in W'}$ (which may be different to the keys received in the garbling phase), and defines the adversary's input as follows.

- INPUT EXTRACTION. For each input wire $w \in W'$, the simulator computes $\rho_w = \Lambda_w \oplus \lambda_w$ and fixes the adversary's input $\{\vec{x}_I\}$ to be the concatenation of these bits. $\mathscr{S}$ sends this input to the trusted party computing f, receiving the output $\boldsymbol{y} = (y_1, \ldots, y_m)$. Note that $\mathscr{A}$ may still provide inconsistent input keys with ρ which we view as providing incorrect PRF values for these wires.

5. SIMULATED GARBLED CIRCUIT GENERATION. Upon receiving the adversary's (`OpenGarbling`) message on $\mathscr{F}_{\text{Preprocessing}}$ the simulator completes the generation of the garbled circuit as follows.

   - It first generates the honest parties' keys $\{k^i_{w,\Lambda_w}\}_{i\in\bar{I},w\in W}$ associated with every internal wire $w \in W$ that is an output of an AND gate. Note that for the honest parties the simulator generates a single key per wire.
   - Next, the simulator chooses a random $\Lambda_w \leftarrow \{0,1\}$ for the public value on every internal wire $w \in W$ that is an output of an AND gate, except for the circuit output wires. For the t-th output wire, $\mathscr{S}$ defines $\Lambda_w = \lambda_w \oplus y_t$ (recall that the masking values for the output wires are already fixed at this point, so the external values must be consistent with the output $\boldsymbol{y} = y_1, \ldots y_m$).
   - For every XOR gate with input wires u and v and output wire w, $\mathscr{S}$ sets $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus \boldsymbol{r}^i$ for all $i \in [n]$, and $\Lambda_w = \Lambda_u \oplus \Lambda_v$.
   - ACTIVE PATH GENERATION. In the next step the simulator computes an active path of the garbled circuit which corresponds to the sequence of keys that will be observed by the adversary. More formally, for every AND gate g that is not an output gate, $\mathscr{S}$ honestly generates the entry in row (Λ_u, Λ_v), where Λ_u (resp. Λ_v) is the public value associated to the left (resp. right) input wire to g. Namely, the simulator computes

   $$\tilde{g}^j_{\Lambda_u,\Lambda_v} = \left(\bigoplus_{i=1}^{n} F_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus k^j_{w,\Lambda_w}$$

   fixing $\tilde{\mathbf{g}}_{\Lambda_u,\Lambda_v} = \tilde{g}^1_{\Lambda_u,\Lambda_v} \circ \ldots \circ \tilde{g}^n_{\Lambda_u,\Lambda_v}$. The remaining three rows are sampled uniformly at random from $\{0,1\}^{n\kappa}$. Importantly, $\mathscr{S}$ never uses the inactive keys $k^i_{u,\bar{\Lambda}_u}, k^i_{v,\bar{\Lambda}_v}$ and $k^i_{w,\bar{\Lambda}_w}$ in order to generate the garbled circuit.

6. The simulator hands the adversary the complete garbled circuit. In case the adversary aborts, the simulator sends $\perp$ to the trusted party and aborts. Otherwise, the simulator obtains an additive error $\boldsymbol{e} = \{e^{a,b}_g\}_{a,b\in\{0,1\},g\in G}$ and computes the modified garbled circuit as $\tilde{\mathbf{g}}_{\Lambda_u,\Lambda_v} + e^{\Lambda_u,\Lambda_v}_g$.

Next, the simulator evaluates the modified circuit using the input wire keys $\{\hat{k}^i_{w,\Lambda_w}\}_{w\in W',i\in I}$ and $\{k^j_{w,\Lambda_w}\}_{w\in W'',j\in\bar{I}}$ and checks whether the honest parties would have aborted. Namely, for each gate, whether the evaluation reveals the honest parties' keys associated with Λ_w for w the output wire of some AND gate. If there exists a gate for which the evaluation does not yield the key associated with Λ_w then the simulator outputs fail and aborts.

This concludes the description of the simulation. Note that the difference between the simulated and the real executions is the way the garbled circuit is generated. More concretely, the simulated garbled circuit is only generated *after* the simulator extracts the adversary's input. Moreover, the simulated garbled gates include a single row that is properly produced, whereas the remaining three rows are picked at random. Let $\mathbf{HYB}^{\mathscr{F}_{\text{Preprocessing}}}_{\Pi_{\text{BMR}},\mathscr{A},\mathscr{Z}}(1^\kappa,z)$ denote the output distribution of the adversary $\mathscr{A}$ and honest parties in a real execution using Π_{BMR} with adversary $\mathscr{A}$. Moreover, let $\mathbf{IDEAL}_{\mathscr{F},\mathscr{S},\mathscr{Z}}(1^\kappa,z)$ denote the output distribution of $\mathscr{S}$ and the honest parties in an ideal execution.

We next define Flip to be the event that there exists an AND gate g and an honest party P_j, who, when evaluating the modified garbled circuit, (1) Does not abort; and (2) Obtains the incorrect key $k^j_{w,\bar{\Lambda}_w}$ for the output wire w of g. Note that this event implies that the adversary causes P_j to compute an incorrect value, as the bit value being transferred within the output wire w is now flipped with respect to P_j.

To prove that this event occurs with negligible probability, we consider an execution $\widetilde{\mathbf{IDEAL}}$ where a simulator $\tilde{\mathscr{S}}$ produces a view that is identical to the view produced by $\mathscr{S}$ in **IDEAL**. Namely, the adversary's view is simulated exactly as in **IDEAL** by a simulator $\tilde{\mathscr{S}}$ with the exception that $\tilde{\mathscr{S}}$ further picks the global differences $\{\boldsymbol{r}^j\}_{j\in\bar{I}}$ and computes inactive keys $k^j_{w,\bar{\Lambda}_w} = k^j_{w,\Lambda_w}\oplus\boldsymbol{r}^j$ for $j\in\bar{I},w\in W$ (which are never defined by $\mathscr{S}$). Moreover, the event for which the simulator outputs fail and aborts is modified as follows. Namely, the simulator aborts if there exists a gate for which the evaluation does not yield the key associated with Λ_w or with $\bar{\Lambda}_w$. Note that this event is well-defined since all the wire keys are chosen in this game. Then the difference between **IDEAL** and $\widetilde{\mathbf{IDEAL}}$ is the event that $\mathscr{S}$ aborts whereas $\tilde{\mathscr{S}}$ does not abort. Note that this event occurs when the adversary successfully flipped a wire value. Specifically, this will yield a valid evaluation in $\widetilde{\mathbf{IDEAL}}$ but not in **IDEAL**. We next show that this event occurs with negligible probability.

Fix the additive error $\boldsymbol{e}=\{e^{a,b}_g\}_{a,b\in\{0,1\},g\in G}$ that is added to the simulated garbled circuit, and let $\tilde{\mathbf{g}}_{a,b}+e^{a,b}_g$ for all $g\in G$ and $a,b\in\{0,1\}$ denote the garbled circuit with the additive error. We prove that Flip only occurs with negligible probability in $\widetilde{\mathbf{IDEAL}}$ which implies that the statement also holds in **IDEAL**. Intuitively, this is due to the fact that the adversary can only succeed in this attack by guessing correctly the global difference $\boldsymbol{r}^j$ of a honest party.

Lemma 4.5.1. *The probability that* Flip *occurs in* $\widetilde{\mathbf{IDEAL}}$ *is no more than* $2^{-\kappa}$.

Proof: Recall first that the simulated garbling in **IDEAL** involves only generating a single key k_w^j per wire and per honest party, which either corresponds to $k_{w,0}^j$ or $k_{w,1}^j$. Consequently, the simulator does not even need to choose a global difference $\boldsymbol{r}^i$ in order to complete the garbling. Furthermore, the simulator in $\widetilde{\textbf{IDEAL}}$ does generate these extra values, but never uses them; this means that the simulated garbled circuit given to $\mathscr{A}$ is completely independent of the honest parties' global differences.

Now, suppose Flip occurs with respect to party P_j, and let g be the first flipped AND gate (in some topological order), with input wires u, v and output wire w. Then, because P_j did not abort, the keys on wires u and v obtained by P_j must contain $k_{u,\Lambda_u}^j, k_{v,\Lambda_v}^j$. Note that it is possible that the corrupt parties' keys for these wires may be incorrect, so we denote these by $\hat{k}_u^i, \hat{k}_w^i$, for $i \in I$. Since gate g was flipped, the j-th entry of the active row of the garbled gate is

$$\hat{g}^j_{\Lambda_u,\Lambda_v} = \bigoplus_{i\in\bar{I}}^{n}\left(F_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus \bigoplus_{i\in I}^{n}\left(F_{\hat{k}^i_u,\hat{k}^i_v}(g\|j)\right) \oplus k^j_{w,\bar{\Lambda}_w}.$$

To cause this to happen, the adversary needs to introduce an error into this entry of the original garbled gate $\tilde{\boldsymbol{g}}$, given by:

$$\begin{aligned}\Delta_g &:= \hat{g}^j_{\Lambda_u,\Lambda_v} \oplus \tilde{g}^j_{\Lambda_u,\Lambda_v} \\ &= \bigoplus_{i\in I}^{n}\left(F_{\hat{k}^i_u,\hat{k}^i_v}(g\|j) \oplus F_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus \boldsymbol{r}^j\end{aligned}$$

This boils down to correctly guessing $\boldsymbol{r}^j$ for the honest party P_j, which is bounded by $2^{-\kappa}$ as $\boldsymbol{r}^j$ is picked truly at random and independently of all other items in the execution. □

In the next step we prove that the ideal and real executions are indistinguishable, conditioned on the event Flip not occurring.

Lemma 4.5.2. *Conditioned on the event* $\overline{\text{Flip}}$*, the following two distributions are computationally indistinguishable:*

- $\{\textbf{HYB}^{\mathscr{F}_{\text{Preprocessing}}}_{\Pi_{\text{BMR}},\mathscr{A},\mathscr{Z}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$
- $\{\textbf{IDEAL}_{\mathscr{F},\mathscr{S},\mathscr{Z}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$

Proof: We begin by defining a slightly modified real execution $\widetilde{\textbf{HYB}}$, where the creation of the garbled circuit is moved from the preprocessing stage to the online computation stage, after the parties have broadcast their masked inputs. Moreover, the generation of the garbled circuit is modified so that upon receiving the honest parties' inputs $\{\rho_i\}_{i\in\bar{I}}$ and extracting the corrupted parties' inputs $\{\rho_i\}_{i\in I}$, the simulator $\widetilde{\mathscr{S}}$ first evaluates the circuit C_f, computing the actual bit ℓ_w to be transferred through each $w \in W$, where W is the set of wires of C_f. It then chooses two keys $k^i_{w,0}, k^i_{w,1}$ and a random bit λ^i_w for all $i \in \bar{I}$ and $w \in W$, and fixes the active key for this wire to be $(k^1_{w,\ell_w\oplus\lambda_w}, \ldots, k^n_{w,\ell_w\oplus\lambda_w})$. The rest of this hybrid is identical to the real execution. Note

that the garbled circuit is still computed according to $\mathscr{F}_{\text{Preprocessing}}$, and the rest of the protocol is identical to **HYB**, which induces the same view for the adversary. This hybrid execution is needed in order to construct a distinguisher for the correlation robustness assumption. Let $\widetilde{\mathbf{HYB}}^{\mathscr{F}_{\text{Preprocessing}}}_{\Pi_{\text{BMR}},\mathscr{A},\mathscr{Z}}(1^\kappa, z)$ denote the output distribution of the adversary $\mathscr{A}$ and honest parties in this game.

Our proof of the lemma follows by a reduction to the correlation robustness of the PRF F (cf. Definition 4.1). Assume by contradiction the existence of an environment $\mathscr{Z}$, an adversary $\mathscr{A}$ and a non-negligible function $p(\cdot)$ such that

$$\left|\Pr[\mathscr{Z}(\widetilde{\mathbf{HYB}}^{\mathscr{F}_{\text{Preprocessing}}}_{\Pi_{\text{BMR}},\mathscr{A},\mathscr{Z}}(1^\kappa, z)) = 1] - \Pr[\mathscr{Z}(\mathbf{IDEAL}_{\mathscr{F},\mathscr{S},\mathscr{Z}}(1^\kappa, z)) = 1]\right| \geq \frac{1}{p(\kappa)}$$

for infinitely many κ's. We construct a distinguisher $\mathscr{D}'$ with access to an oracle $\mathscr{O}$ (that implements either Circ or Rand) that breaks the security of the correlation robustness assumption. Namely, we show that

$$\left|\Pr[\boldsymbol{r} \leftarrow \{0,1\}^n; \mathscr{A}^{\mathsf{Circ}_{\boldsymbol{r}}(\cdot)}(1^\kappa) = 1] - \Pr[\mathscr{A}^{\mathsf{Rand}(\cdot)}(1^\kappa) = 1]\right| \geq \frac{1}{p(\kappa)}.$$

Distinguisher $\mathscr{D}'$ receives the environment's input z and internally invokes $\mathscr{Z}$ and simulator $\mathscr{S}$, playing the role of functionality f. In more details,

- $\mathscr{D}'$ internally invokes $\mathscr{Z}$ that fixes the honest parties' inputs $\boldsymbol{\rho}$.
- $\mathscr{D}'$ emulates the communication with the adversary (controlled by $\mathscr{Z}$) in the initialization, preprocessing and garbling steps as in the simulation with $\mathscr{S}$.
- For each wire u, let $\ell_u \in \{0,1\}$ be the actual value on wire u. Note that these values, as well as the output of the computation y, can be determined since $\mathscr{D}'$ knows the actual input of all parties to the circuit (where the adversary's input is extracted as in the simulation with $\mathscr{S}$).
- It next constructs the garbled circuit as follows. For each wire w in the circuit that is an output wire of an AND gate and $i \in \bar{I}$, it samples a key k^i_w and a public value Λ_w. Using the internal values ℓ_w, we can also compute the masks $\lambda_w = \ell_w \oplus \Lambda_w$.
- For each wire that is the output of an XOR gate with input wires u and v and output wire w, the distinguisher sets $k^i_w = k^i_u \oplus k^i_v$ for all $i \in [n]$, and $\Lambda_w = \Lambda_u \oplus \Lambda_v$.
- The distinguisher picks an honest party, say P_{i_0}, and samples global differences $\boldsymbol{r}^i$ for $i \in \bar{I} \setminus \{i_0\}$. For every $i \in \bar{I} \setminus \{i_0\}$ and $w \in W$, $\mathscr{D}'$ now has both keys $k^i_{w,\Lambda_w} = k^i_w$ and $k^i_{w,\overline{\Lambda}_w} = k^i_w \oplus \boldsymbol{r}^i$.
- Finally, for each wire that is the output of an AND gate g with input wires u and v and output wire w, the distinguisher computes four ciphertexts c_{00}, c_{01}, c_{10} and c_{11} as the garbled gate, that are generated as follows,

- First, the j-th entry in the (Λ_u, Λ_v)-th row is computed as

$$\left(\bigoplus_{i=1}^{n} F_{k^i_{u,\Lambda_u}, k^i_{v,\Lambda_v}}(g \| j)\right) \oplus k^j_{w,\Lambda_w}.$$

- Next, for all $(a,b) \in \{0,1\}^2$ such that $(a,b) \neq (\Lambda_u, \Lambda_v)$ the distinguisher sets $\ell_{a,b} = 0$ if $g(a \oplus \lambda_u, b \oplus \lambda_v) = \ell_w$, and sets $\ell_{a,b} = 1$ otherwise. It then queries $h^j_{a,b} = \mathcal{O}(k^{i_0}_{u,\Lambda_u}, k^{i_0}_{v,\Lambda_v}, g, j, a \oplus \lambda_u, b \oplus \lambda_v, \ell_{a,b})$, and sets the j-th entry of row (a,b) in the garbled gate to be

$$\left(\bigoplus_{i \neq i_0} F_{k^i_{u,a}, k^i_{v,b}}(g \| j)\right) \oplus h^j_{a,b} \oplus k^j_{w,\Lambda_w}.$$

- For the output wires the distinguisher sets the external values as in the simulation.

- $\mathcal{D}'$ hands the adversary the complete description of the garbled circuit and concludes the execution as in the simulation with $\mathcal{S}$.

- $\mathcal{D}'$ outputs whatever $\mathcal{Z}$ does.

Note first that $\mathcal{D}'$ only makes legal queries to its oracle. Furthermore, if $\mathcal{O} = \mathsf{Circ}$ then the view of $\mathcal{A}$ is identically distributed to its view in the real execution of the protocol on the given inputs, whereas if $\mathcal{O} = \mathsf{Rand}$ then $\mathcal{A}$'s view is distributed identically to the output of the simulator described previously since the oracle's response is truly random in this case. This completes the proof. □

Finally, we demonstrate that the probability Flip occurs in **HYB** is negligible as well due to indistinguishability of executions. This concludes the proof as it demonstrates that with overwhelming probability the adversary is getting caught whenever cheating in the computation of the PRF values.

Lemma 4.5.3. *The probability that* Flip *occurs in* **HYB** *is bounded by* $2^{-\kappa} + \mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$.

Proof: Intuitively speaking, we prove that if Flip occurs in the real execution with a non-negligible probability, then we can leverage this distinguishing gap in order to break the correlation robustness assumption. Namely, if this event occurs then it is possible to extract $\boldsymbol{r}^i$ and all pairs of inputs keys associated with every wire with respect to an honest party. Given all keys it is possible to recompute the garbled circuit and verify whether it was generated honestly or as in the simulation. More formally, assume by contradiction that

$$\Pr[\mathsf{Flip} \text{ occurs in } \mathbf{HYB}] \geq \frac{1}{q(\kappa)}$$

for some non-negligible function $q(\cdot)$ and infinitely many κ's. We construct a distinguisher $\mathcal{D}$ that breaks the security of the underlying correlation robust PRF with non-negligible probability as follows.

1. Distinguisher $\mathscr{D}$ is identically defined as the distinguisher in the proof of Lemma 4.5.2, externally communicating with an oracle $\mathscr{Q}$ that either realizes the function Circ or Rand, while internally invoking $\mathscr{A}$. The only difference is that $\mathscr{D}$ chooses i_0 at random. This is due to the fact that the event Flip holds with respect to (at least) one honest party, whose identity is unknown.

2. Upon receiving the modified garbled circuit from $\mathscr{A}$, $\mathscr{D}$ evaluates the circuit on the parties' inputs and compares every active key $\tilde{k}_w^i$ that is revealed during the execution with the actual active key k_w^i that was created by $\mathscr{D}$ in the garbling phase. For every gate g for which there exists a difference $\Delta_g^i = \tilde{k}_w^i \oplus k_w^i$ for all $i \in \bar{I}$, $\mathscr{D}$ sets $\boldsymbol{r}_g^i = \Delta_g^i$.

3. For every gate g for which $\mathscr{D}$ recorded a global difference $\boldsymbol{r}_g^{i_0}$ for party i_0, it defines the inactive key for the output wire $w \in W$ of this gate by $k_{w,\bar{\Lambda}_w}^{i_0} = k_{w,\Lambda_w}^{i_0} \oplus \boldsymbol{r}_g^{i_0}$. Next, for some gate g' for which wire w is an input wire (say associated with left input wire to g' w.l.o.g., $\mathscr{D}$ queries its oracle on $(k_{w,\bar{\Lambda}_w}^{i_0}, k_v^{i_0}, g, j, \bar{a} \oplus \lambda_w, b \oplus \lambda_v, \ell_{a,b})$ where $k_v^{i_0}$ is the active key associated with the other input wire of P_{i_0}. $\mathscr{D}$ compares this outcome with the values it obtained from its oracle for the query $(k_{w,\Lambda_w}^{i_0}, k_v^{i_0}, g, j, a \oplus \lambda_w, b \oplus \lambda_v, \ell_{a,b})$. If equality holds, then $\mathscr{D}$ outputs Circ.

4. Upon concluding the execution so that $\mathscr{D}$ did not output Circ, it returns Rand.

Clearly, whenever Flip occurs with respect to i_0 then $\mathscr{D}$ can identify this event by extracting some inactive key and querying its oracle in this key. Therefore $\mathscr{D}$ outputs Circ with probability $\frac{1}{q(\kappa)\cdot n}$. On the other hand, due to the claim made in Lemma 4.5.2, Flip rarely occurs in **IDEAL** and thus $\mathscr{D}$ outputs Rand on the event of Flip only with negligible probability. This implies a non-negligible gap with respect to the event occurring in the two executions and concludes the proof. □ ■

4.5.1 The Online Phase with $\mathscr{F}_{\text{Preprocessing}}^{\text{KQ}}$

In this section we now prove the following theorem, for the online protocol based on $\mathscr{F}_{\text{Preprocessing}}$ with key queries, denoted by $\mathscr{F}_{\text{Preprocessing}}^{\text{KQ}}$ and formally defined in Section 4.4.3.

Theorem 4.5.2. *Let f be an n-party functionality $\{0,1\}^{n\kappa} \mapsto \{0,1\}^{\kappa}$ and assume that F is a circular 2-correlation robust PRF. Then Protocol Π_{BMR}, from Figure 4.11, UC-securely computes f in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\mathscr{F}_{\text{Preprocessing}}^{\text{KQ}}$-hybrid model.*

In what follows, we discuss how to adapt the proof of Theorem 4.5.1 to support key queries.

Proof Sketch: We first modify the simulator specified in that proof. Namely, upon receiving the adversary's queries (i, R'), the simulator outputs 0. Intuitively, we claim that with overwhelming probability the adversary only sees zero responses to its key queries as it can only guess a global key with negligible probability. In the following, we formalize this intuition and discuss how

to modify the proof of Theorem 4.5.1 by re-proving Lemma 4.5.1. Namely, we need to take into account the fact that the probability that the event Flip occurs also depends on the leakage obtained by the key queries made to the functionality.

We define a new hybrid game **H** where the simulator $\mathcal{S}_{\mathbf{H}}$ is defined identically to simulator $\mathcal{S}$ with the exception that $\mathcal{S}_{\mathbf{H}}$ knows the honest parties' inputs and further generates the inactive keys by picking global differences for the honest parties. Furthermore, for every key query (i, R') made by $\mathcal{A}$, $\mathcal{S}_{\mathbf{H}}$ verifies first whether $R' = R^i$ and aborts in case equality holds. Otherwise, it replies with 0. Nevertheless, $\mathcal{S}_{\mathbf{H}}$ garbles the circuit the same way $\mathcal{S}$ does. Let Guess denote the event in **H** for which the adversary makes a key query R^i (meaning, it guesses the correct R^i value). We prove the following.

Lemma 4.5.4. *The probability that* Flip *occurs in* **H** *is no more than* $(q+1)/2^\kappa$*, where* q *is the number of key queries.*

Proof: We analyze the probability that the event Flip occurs.

$$\begin{aligned}\Pr(\mathsf{Flip}) &= \Pr(\mathsf{Flip}|\mathsf{Guess}) \cdot \Pr(\mathsf{Guess}) + \Pr(\mathsf{Flip}|\overline{\mathsf{Guess}}) \cdot \Pr(\overline{\mathsf{Guess}}) \\ &\leq \Pr(\mathsf{Guess}) + \Pr(\mathsf{Flip}|\overline{\mathsf{Guess}}) \leq q/2^\kappa + 2^{-\kappa}.\end{aligned}$$

□

This implies that the distributions induced within **IDEAL** and **H** are statistically close since the only difference is whenever event Guess occurs. We next claim that the proof of Lemma 4.5.2 holds with respect to **H** and **HYB**.

Lemma 4.5.5. *Conditioned on the event* $\overline{\mathsf{Flip}}$*, the following two distributions are computationally indistinguishable:*

- $\{\mathbf{HYB}^{\mathcal{F}_{\text{Preprocessing}}^{KQ}}_{\Pi_{\text{BMR}},\mathcal{A}}(1^\kappa, z)\}_{\kappa\in\mathbb{N}, z\in\{0,1\}^*}$
- $\{\mathbf{H}^{\mathcal{F}_{\text{Preprocessing}}^{KQ}}_{\Pi_{\text{BMR}},\mathcal{S}_{\mathbf{H}}}(1^\kappa, z)\}_{\kappa\in\mathbb{N}, z\in\{0,1\}^*}$

Proof: This proof will have to incorporate the key queries as well. Namely, distinguisher $\mathcal{D}'$ first picks the identity of party i_0 at random. Then, whenever a key query (i_0, R^{i_0}) is made by $\mathcal{A}$, $\mathcal{D}'$ uses it to calculate the inactive keys of party P_{i_0} and checks whether this yields the garbling it obtained from its oracle. In case it does, $\mathcal{D}$ outputs Circ. Otherwise, it outputs Rand. Otherwise, if at the end of the execution no queries have resulted in a correct garbling, output Rand. □

Finally, we reprove Lemma 4.5.3 by demonstrating that if the success probability of Flip is non-negligibly higher in **HYB**, then we can distinguish the two executions **HYB** and **H**.

Lemma 4.5.6. *The probability that* Flip *occurs in* **HYB** *is bounded by* $2^{-\kappa} + \mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$.

Proof: This proof follows similarly to the proof of Lemma 4.5.3 in the sense that the reduction additionally needs to reply the adversary's key queries. Namely, for each query (i, R^i) such that $i \neq i_0$, $\mathscr{D}$ can answer this query as it picked the global difference for that party. Moreover, for each query (i_0, R^{i_0}), $\mathscr{D}$ uses this query to fix a set of inactive keys for party i_0 and verifies whether this guess is correct by recomputing the garbled circuit and comparing it with the original garbled circuit. If equality holds then $\mathscr{D}$ responses with 1 to this query. The rest of the proof follows identically. □ ■

4.6 Performance

In this section we present implementation results for our protocol from Section 4.4 for up to 9 parties. We also analyse the concrete communication complexity of the protocol and compare this with previous, state-of-the-art protocols in a similar setting.

We have made a couple of tweaks to our protocol to simplify the implementation. We moved the **Open Garbling** stage to the preprocessing phase, instead of the online phase. This optimizes the online phase so that the amount of communication is independent of the size of the circuit. This change means that our standard model security proof would no longer apply, but we could prove it secure using a random oracle instead of the circular 2-correlation robust PRF, similarly to [21, 95]. Secondly, when not working in a modular fashion with a separate preprocessing functionality, the share rerandomization step in the output stage is not necessary to prove security of the entire protocol, so we omit this.

4.6.1 Implementation

Note: The following implementation work was not carried out by the author, but by Assi Barak, Moriya Farbstein and Lior Koskas from the software team at Bar-Ilan University. It appears here for completeness and a better understanding of the previous results in this chapter.

We implemented our variant of the multi-party TinyOT protocol (Section 4.7) using the `libOTe` library [117] for the fixed-correlation OTs, and tested it for between 3 and 9 parties. We benchmarked the protocol over a 1Gbps LAN on 5 servers with 2.3GHz Intel Xeon CPUs with 20 cores. For the experiments with more than 5 parties, we had to run more than one party per machine; this should not make much difference in a LAN, as the number of threads being used was still fewer than the number of cores. As benchmarks, we measured the time for securely computing the circuits for AES (6800 AND gates) and SHA-256 (90825 AND gates).

For the TinyOT bit and triple generation, every pair of parties needs two correlated OT instances running between them (one in each direction). We ran each OT instance in a separate thread with `libOTe`, so that each party uses $2(n-1)$ OT threads. This gave a small improvement ($\approx 6\%$) compared with running $n-1$ threads. We also considered a multiple execution setting, where many (possibly different) secure computations are evaluated. Provided the total number of

	AES ($B = 5$)	AES ($B = 3$)	SHA-256 ($B = 5$)	SHA-256 ($B = 3$)
Prep.	1329	586.9	10443	6652
Online	35.34	33.30	260.58	252.8

Table 4.2: Runtimes in ms for AES and SHA-256 evalution with 9 parties.

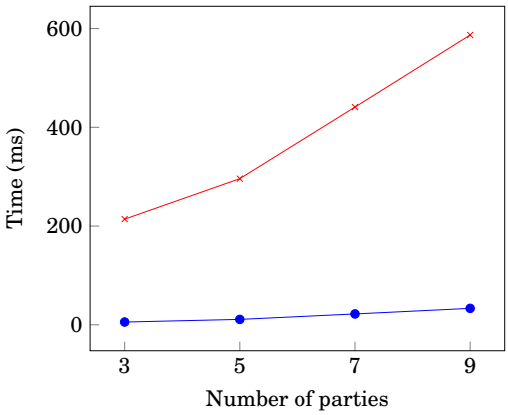


Figure 4.12: AES performance (6800 AND gates).

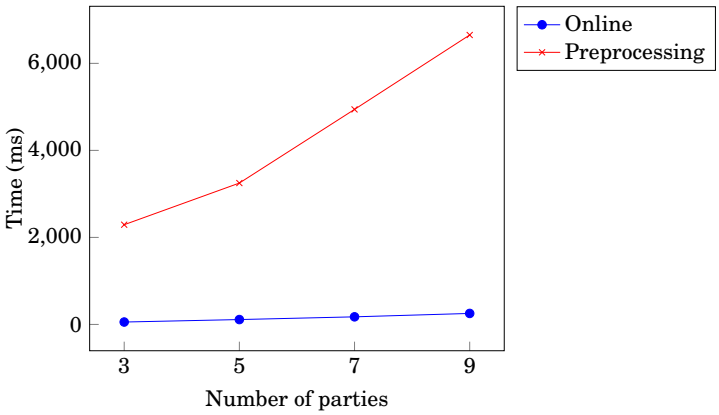


Figure 4.13: SHA-256 performance (90825 AND gates).

AND gates in the circuits being evaluated is at least 2^{20}, this allows us to generate the TinyOT triples for all executions at once using a bucket size of $B = 3$, compared with $B = 5$ for one execution of AES or $B = 4$ for one execution of SHA-256. Since the protocol in Section 4.7 scales with B^2, this has a big impact on performance. The results for 9 parties, for the different choices of B, are shown in Table 4.2.

Figures 4.12–4.13 show how the performance of AES and SHA-256 scales with different numbers of parties, in the amortized setting. Although the asymptotic complexity is quadratic, the runtimes grow relatively slowly as the number of parties increases. This is because in the preprocessing phase, the amount of data sent *per party* is actually linear. However, the super-linear trend is probably due to the limitations of the total network capacity, and the computational costs.

4.6.1.1 Comparison with other works.

We calculated the cost of computing the SPDZ-BMR protocol [93] using [80] to derive estimates for creating the SPDZ triples (the main cost). Using MASCOT over $\mathbb{F}_{2^\kappa}$ with Free-XOR, SPDZ-BMR requires $3n + 1$ multiplications per garbled AND gate. This gives an estimated cost of at least 14 seconds to evaluate AES, which is over 20x slower than our protocol.

At the time of writing this thesis, the only other implementation of actively secure, constant-round, dishonest majority MPC is the work of [129], which presents implementation figures for

up to 256 parties running on Amazon servers. Their runtimes with 9 parties in a LAN setting are around 200ms for AES and 2200ms for SHA-256, which is around 3 times faster than our results. However, their LAN setup has 10Gbps bandwidth, whereas we only tested on machines with 1Gbps bandwidth. Since the bottleneck in our implementation is mostly communication, it seems that our implementation could perform similar to or even faster than theirs in the same environment, despite our higher communication costs. However, it is not possible to make an accurate comparison without testing both implementations in the same environment.

Our implementation does not scale well to a WAN environment in the cloud because we have not fully exploited the low round complexity of the protocol. However, the LAN results in our work serve to demonstrate the practicality of the protocol and its low round complexity means that it will still be practical in a WAN setting, even more with some refactoring.

Compared with protocols based solely on secret-sharing, such as SPDZ and TinyOT, the advantage of our protocol is the low round complexity. We have not yet managed to benchmark our protocol in a WAN setting, but since our total round complexity is less than 20, it should perform reasonably well. With secret-sharing, using e.g. TinyOT, evaluating the AES circuit requires at least 40 rounds in just the online phase (it can be done with 10 rounds [46], but this uses a special representation of the AES function, rather than a general circuit), whilst computing the SHA-256 circuit requires *4000 rounds*. In a network with 100ms delay between parties, the AES online time alone would be at least 4 seconds, whilst SHA-256 would take over *10 minutes* to securely compute in that setting. If our protocol is run in this setting, we should be able to compute both AES and SHA-256 in just a few seconds (assuming that latency rather than bandwidth is the bottleneck).

4.6.2 Communication Complexity Analysis

We now focus on analysing the concrete communication complexity of the optimized variant of our protocol and compare it with the state of the art in constant-round two-party and multi-party computation protocols. We have not implemented our protocol, but since the underlying computational primitives are very simple, the communication cost will be the overall bottleneck. As a benchmark, we estimate the cost of securely computing the AES circuit (6800 AND gates, 25124 XOR gates), where we assume that one party provides a 128-bit plaintext or ciphertext and the rest of them have an XOR sharing of a 128-bit AES key. This implies we have $128 \cdot n$ input wires and an additional layer of XOR gates in the circuit to add the key shares together. We consider a single set of 128 output wires, containing the final encrypted or decrypted message.

4.6.2.1 Complexity of Our TinyOT-Based Protocol

We now measure the exact communication cost of our optimized protocol based on TinyOT (in the random oracle model), in terms of number of bits sent over the network per party (multiply this by n for the overall complexity). We exclude one-time costs such as checking MACs, which can be

done in a batch at the end, and initializing the base OTs. Consider a circuit with G AND gates, I input wires (in total) and O output wires. The costs of the different stages are as follows, with computational security parameter $\kappa = 128$ and statistical security parameter $s = 40$.

TINYOT PREPROCESSING: One triple and one random bit per AND gate, plus one random bit per input wire. From the analysis in Section 4.7 this gives

$$(504B^2 + 168)(n-1)G + 168(n-1)I.$$

GARBLING: Two bit openings for each AND gate (for the bit multiplication), one bit opening for every output wire and one private opening for every input wire, gives

$$2(n-1)G + (n-1)O + (n-1)I.$$

OPEN GARBLING: The rerandomization step costs $\kappa(n-1)$ bits per party. Opening the garbled circuit can be done efficiently by each party sending their share to P_1, who broadcasts the result; this costs $4n\kappa G$ bits per party, giving a total of

$$4n\kappa G + \kappa(n-1).$$

ONLINE: If party P_i has I_i input bits then the cost for P_i is $I_i + I \cdot (n-1) \cdot \kappa$ bits.

Note that for a single execution of AES we have $G = 6800, I = 128n$ and $O = 128$, which means for multi-party TinyOT we can choose $B = 4$, following the combinatorial analysis of [59].

4.6.2.2 Two Parties

In Table 4.3 we compare the cost of our protocol in the two-party case, with state-of-the-art secure two-party computation protocols. We instantiate our TinyOT-based preprocessing method with the optimized, two-party TinyOT protocol from [128], lowering the previous costs further. For consistency with the other two-party protocols, we divide the protocol costs into three phases: function-independent preprocessing, which only depends on the size of the circuit; function-dependent preprocessing, which depends on the exact structure of the circuit; and the online phase, which depends on the parties' inputs. As with the implementation, we move the garbled circuit opening to the function-dependent preprocessing, to simplify the online phase.

The online phase of the modified protocol is just two rounds of interaction, and has the lowest online cost of *any* actively secure two-party protocol.[3] The main cost of the function-dependent preprocessing is opening the garbled circuit, which requires each party to send 8κ bits per AND gate. This is slightly larger than the best Yao-based protocols, due to the need for a set of keys for every party in BMR.

[3] If counting the *total* amount of data sent, in both directions, our online cost would be larger than [128], which is highly asymmetric. In practice, however, the latency depends on the largest amount of communication from any one party, which is why we measure in this way.

Protocol	# Executions	Function-indep. prep.	Function-dep. prep.	Online
[118]	32	–	3.75 MB	25.76 kB
	128	–	2.5 MB	21.31 kB
	1024	–	1.56 MB	16.95 kB
[107]	1	14.94 MB	227 kB	16.13 kB
	32	8.74 MB	227 kB	16.13 kB
	128	7.22 MB	227 kB	16.13 kB
	1024	6.42 MB	227 kB	16.13 kB
[128]	1	2.86 MB	570 kB	4.86 kB
	32	2.64 MB	570 kB	4.86 kB
	128	2.0 MB	570 kB	4.86 kB
	1024	2.0 MB	570 kB	4.86 kB
Ours + [128]	1	2.86 MB	872 kB	4.22 kB
	32	2.64 MB	872 kB	4.22 kB
	128	2.0 MB	872 kB	4.22 kB
	1024	2.0 MB	872 kB	4.22 kB

Table 4.3: Communication estimates for secure AES evaluation with our protocol and previous works in the two-party setting. Cost is the maximum amount of data sent by any one party, per execution.

In the batch setting, where many executions of the *same circuit* are needed, protocols such as [118] clearly still perform the best. However, if many circuits are required, but they may be different, or not known in advance, then our multi-party protocol is highly competitive with two-party protocols.

4.6.2.3 Comparison with Multi-Party Protocols

In Table 4.4 we compare our work with previous constant-round protocols suitable for any number of parties, again for evaluating the AES circuit. We do not present the communication complexity of the online phase as we expect it to be very similar in all of the protocols. We denote by MASCOT-BMR-FX an optimized variant of [93], modified to use Free-XOR as in our protocol, with multiplications done using the OT-based MASCOT protocol [80].

As in the previous section, we move the cost of opening the garbled circuit to the preprocessing phase for all of the presented protocols (again relying on random oracles). By applying this technique the online phase of our work is just two rounds, and has exactly the same complexity as the current most efficient *semi-honest* constant-round MPC protocol for any number of parties [25], except we achieve active security. We see that with respect to other actively secure protocols, we improve the communication cost of the preprocessing by around 2–4 orders of magnitude. Moreover, our protocol scales much better with n, since the complexity is $O(n^2)$ instead of $O(n^3)$.

Protocol	Security	Function-indep. prep.		Function-dep. prep.	
		$n = 3$	$n = 10$	$n = 3$	$n = 10$
SPDZ-BMR	active	25.77 GB	328.94 GB	61.57 MB	846.73 MB
SPDZ-BMR	covert, pr. $\frac{1}{5}$	7.91 GB	100.98 GB	61.57 MB	846.73 MB
MASCOT-BMR-FX	active	3.83 GB	54.37 GB	12.19 MB	178.25 MB
[129]	active	4.8 MB	20.4 MB	1.3 MB	4.4 MB
Ours	active	14.01 MB	63.22 MB	1.31 MB	4.37 MB

Table 4.4: Comparison of the cost of our protocol with previous constant-round MPC protocols in a range of security models, for secure AES evaluation. Costs are the amount of data sent over the network per party.

The concurrent work of Katz et al. [129] requires around 3 times less communication than our protocol, which is due to their optimized version of the multi-party TinyOT protocol.

4.7 A Multi-Party TinyOT-Style Protocol

Here we describe the full protocol for realizing the $\mathscr{F}_{\text{n-TinyOT}}$ functionality. It essentially consists of the bit triple generation protocol from [58], with some minor modifications, and a method for producing random shared bits with a consistency check that is similar to the bit/string check from Section 4.3.3.

We first recall the two-party and n-party MAC representations from Section 4.4:

$$[x^i]_{i,j} = (x^i, M^i_j, K^j_i)$$
$$[x] = (x^i, \{M^i_j, K^i_j\}_{j\neq i})_{i\in[n]}, \quad M^i_j = K^j_i + x^i \cdot R^j$$

where in the two-party sharing $[x^i]_{i,j}$, P_i holds the share x^i and MAC M^i_j, whilst P_j holds the local key K^j_i and a fixed, global key R^j. In the n-party sharing, each party P_i holds $n-1$ MACs on x^i, as well as a key on x^j, for each $j \neq i$, and a global key R^i. Note that if P_i holds x^i and P_j holds the key R^j, a sharing $[x^i]_{i,j}$ can easily be created using one call to the correlated OT functionality (Figure 4.3), in which the correlation R^j is fixed by P_j in the initialization stage.

As required in the modified preprocessing protocol from Section 4.4, we need a method for opening $[x]$-shared values, both to all parties, and privately to a single party. These are straightforward, shown in Figure 4.14–4.15.

The main protocol, shown in Figure 4.16, consists of two main parts, for creating shared random bits, and for multiplication (AND) triples. Creating shared bits is straightforward, by

Subprotocol Π_{Open}

To open a shared value $[x]$ to all parties:

1. Each party P_i broadcasts its share x^i, and sends the MAC M_j^i to P_j, for $j \neq i$.
2. All parties compute $x = x^1 + \cdots + x^n$.
3. Each P_i has received MACs M_i^j, for $j \neq i$, and checks that

$$M_i^j = K_j^i + x^j \cdot R^i.$$

If any check fails, broadcast $\perp$ and abort.

Figure 4.14: Subprotocol for opening and checking MACs on n-party authenticated secret shares.

Subprotocol Π_{Open}^j

To open a shared value $[x]$ to only P_j:

1. Each party P_i, for $i \neq j$, privately sends its share x^i and MAC M_j^i to P_j.
2. P_j computes $x = x^1 + \cdots + x^n$, and checks that, for each $i \neq j$

$$M_j^i = K_i^j + x^i \cdot R^j.$$

If any check fails, broadcast $\perp$ and abort.

Figure 4.15: Subprotocol for private opening to one party.

using $\mathcal{F}_{\Delta\text{-ROT}}$ to MAC random bits, then opening a random linear combination of the MACs to ensure consistency.

To create shares of a random multiplication triple (x, y, z), each party first locally samples shares x^i, y^i, and then uses $\mathcal{F}_{\Delta\text{-ROT}}$ to authenticate these shares. The MAC on a share x^i is used to obtain a sharing of the product of x^i with a random bit $(u_\ell^{i,j} + v_\ell^{i,j})$ known to P_j (using a hash function), and then P_j converts this to a share of $x^i \cdot y^j$ by sending a correction bit in step 3a. This opens up an avenue for cheating, as P_j may send an incorrect correction value to some P_i. This could result in the triple being correct, or, if $x^i = 0$ the triple would still be correct but P_j would learn the bit x^i. These issues are addressed by the bucket-based cut-and-choose procedure in Figure 4.17, which first checks correctness by sacrificing triples, and then removes any potential leakage on the x values by combining several triples together. Note that the complex bucketing procedure is necessary for these steps, as opposed to simple pairwise checks, because with triples over $\mathbb{F}_2$, one pairwise check (or leakage combiner) can only guarantee correctness (or remove leakage) if *at least one* of the two triples is correct (or leakage-free). So, the cut-and-choose and bucketing procedure are done so that each bucket contains at least one good triple, with overwhelming probability.

Protocol $\Pi_{\text{n-TinyOT}}$

Let $\mathsf{H}:\{0,1\}^\kappa \to \{0,1\}$ be a single-bit output hash function, modeled as a random oracle.

Initialize: 1. Each party P_i samples $R^i \leftarrow \{0,1\}^\kappa$.

2. Every ordered pair (P_i,P_j) calls $\mathscr{F}_{\Delta\text{-ROT}}$, where P_i sends $(\texttt{init},R^i)$ and P_j sends $(\texttt{init})$.

Prep: To create m random shared bits $[b_1],\ldots,[b_m]$ do:

1. Each party P_i samples $m+\kappa$ random bits $b^i_1,\ldots,b^i_m,r^i_1,\ldots,r^i_\kappa \leftarrow \{0,1\}$.
2. Every ordered pair (P_i,P_j) calls $\mathscr{F}_{\Delta\text{-ROT}}$, where P_i is receiver and inputs $(\texttt{extend},b^i_1,\ldots,b^i_m,r^i_1,\ldots,r^i_\kappa)$. Using the output, define sharings $[b_1],\ldots,[b_m],[r_1],\ldots,[r_\kappa]$.
3. Check consistency of the $\mathscr{F}_{\Delta\text{-ROT}}$ inputs as follows:
   a) Call $\mathscr{F}_{\text{Rand}}$ to obtain random field elements $\chi_1,\ldots,\chi_m \in \mathbb{F}_{2^\kappa}$
   b) The parties locally compute (with arithmetic over $\mathbb{F}_{2^\kappa}$)
   $$[C] = \sum_{\ell=1}^{m} \chi_\ell \cdot [b_\ell] + \sum_{h=1}^{\kappa} X^{h-1}\cdot [r_h]$$
   c) Each P_i now has a share $C^i \in \mathbb{F}_{2^\kappa}$, and the MACs and keys $(M^i_j,K^i_j)_{j\neq i}$ from $[C]$.
   d) Each P_i rerandomizes C^i by privately distributing fresh shares, and sums up the shares they receive to obtain a new share $\bar{C}^i$.
   e) Broadcast $\bar{C}^i$ and reconstruct $c = \bar{C}^1+\cdots+\bar{C}^n$.
   f) Each party P_i defines and commits to the $n+1$ values:
   $$C^i, \quad Z^i_j = M^i_j \quad (\text{for } j\neq i), \quad Z^i_i = \sum_{j\neq i} K^i_j + (C+C^i)\cdot R^i.$$
   g) All parties open their commitments and check that, for each $j\in[n]$, $\sum_{i=1}^n Z^i_j = 0$. Additionally, each P_i checks that $Z^j_i = K^i_j + C^j\cdot R^i$. If any check fails, abort.

To create M AND triples, first create $m' = B^2M + c$ triples as follows:

1. Each party P_i samples $x^i_\ell, y^i_\ell \leftarrow \mathbb{F}_2$ for $\ell\in[m']$
2. Every ordered pair (P_i,P_j) calls $\mathscr{F}_{\Delta\text{-ROT}}$, where P_i is receiver and inputs $(\texttt{extend},x^i_1,\ldots,x^i_{m'})$. P_i and P_j obtain their respective value of $[x^i_\ell]_{i,j} = (M^{i,j}_\ell,K^{j,i}_\ell)$, such that $M^{i,j}_\ell = K^{j,i}_\ell + x^i_\ell\cdot R^j \in \mathbb{F}_2^\kappa$.
3. For each $\ell\in[m']$ and each pair of parties (P_i,P_j):
   a) P_j computes $u^{j,i}_\ell = \mathsf{H}(K^{j,i}_\ell)$, $v^{j,i}_\ell = \mathsf{H}(K^{j,i}_\ell + R^j)$, and sends $d = u^{j,i}_\ell + v^{j,i}_\ell + y^j_\ell$ to P_i
   b) P_i computes $w^{i,j}_\ell = \mathsf{H}(M^{i,j}_\ell) + x^i_\ell\cdot d = u^{j,i}_\ell + x^i_\ell\cdot y^j_\ell$
4. Each party P_i defines shares
   $$z^i_\ell = \sum_{j\neq i}(u^{i,j}_\ell + w^{i,j}_\ell) + x^i_\ell\cdot y^i_\ell$$
5. Every ordered pair (P_i,P_j) calls $\mathscr{F}_{\Delta\text{-ROT}}$, where P_i is receiver and inputs $(\texttt{extend},\{y^i_\ell,z^i_\ell\}_{\ell\in[m']})$.
6. Use the above, and the previously obtained MACs on x^i_ℓ, to create sharings $[x_\ell],[y_\ell],[z_\ell]$.

Finally, run $\Pi_{\text{TripleBucketing}}$ on $([x_\ell],[y_\ell],z_\ell])_{\ell\in[m']}$, to output M correct and secure triples.

Figure 4.16: Protocol for TinyOT-style Multi-Party Computation of binary circuits.

Subprotocol $\Pi_{\text{TripleBucketing}}$

The protocol takes as input $m' = B^2 m + c$ triples, which may be incorrect and/or have leakage on the x component, and produces m triples which are guaranteed to be correct and leakage-free.
B determines the bucket size, whilst c determines the amount of cut-and-choose to be performed.

Input: Start with the shared triples $\{[x_i],[y_i],[z_i]\}_{i\in[m']}$.

I: Cut-and-choose: Using $\mathscr{F}_{\text{Rand}}$, the parties select at random c triples, which are opened with Π_{Open} and checked for correctness. If any triple is incorrect, abort.

II: Check correctness: The parties now have $B^2 m$ unopened triples.

1. Use $\mathscr{F}_{\text{Rand}}$ to sample a random permutation on $\{1,\ldots,B^2 m\}$, and randomly assign the triples into mB buckets of size B, accordingly.
2. For each bucket, check correctness of the first triple in the bucket, say $[T] = ([x],[y],[z])$, by performing a pairwise sacrifice between $[T]$ and every other triple in the bucket. Concretely, to check correctness of $[T]$ by sacrificing $[T'] = ([x'],[y'],[z'])$:
   a) Open $d = x + x'$ and $e = y + y'$ using Π_{Open}.
   b) Compute $[f] = [z] + [z'] + d \cdot [y] + e \cdot [x] + d \cdot e$.
   c) Open $[f]$ using Π_{Open} and check that $f = 0$.

III: Remove leakage: Taking the first triple in each bucket from the previous step, the parties are left with Bm triples. They remove any potential leakage on the $[x]$ bits of these as follows:

1. Place the triples into m buckets of size B.
2. For each bucket, combine all B triples into a single triple. Specifically, combine the first triple $([x],[y],[z])$ with $[T'] = ([x'],[y'],[z'])$, for every other triple T' in the bucket:
   a) Open $d = y + y'$ using Π_{Open}.
   b) Compute $[z''] = d \cdot [x'] + [z] + [z']$ and $[x''] = [x] + [x']$.
   c) Output the triple $[x''],[y],[z'']$.

If all the checks and MAC checks passed, output the first triple from each of the m buckets in the final stage.

Figure 4.17: Checking correctness and removing leakage from triples with cut-and-choose.

Subprotocol Π_{Mult}

Given a multiplication triple $[a],[b],[c]$ and two shared values $[x],[y]$, the parties compute a sharing of $x \cdot y$ as follows:

1. Each party broadcasts $d^i = a^i + x^i$ and $e^i = b^i + y^i$.
2. Compute $d = \sum_i d^i$, $e = \sum_i e^i$, and run Π_{Open} to check the MACs on $[d]$ and $[e]$.
3. Output
$$[z] = [c] + d \cdot [b] + e \cdot [a] + d \cdot e = [x \cdot y].$$

Figure 4.18: Subprotocol for multiplying secret shared values using a triple.

4.7.1 Why the Need for Key Queries?

For completeness, we briefly explain why these are needed in $\mathcal{F}_{\text{n-TinyOT}}$, when using this protocol. After the protocol execution the environment learns the honest parties' outputs, which include MAC keys K_i and R. On the other hand, during the protocol the adversary sees values of the form (simplifying things slightly)

$$U = \mathsf{H}(K_i) + \mathsf{H}(K_i + R)$$

where H is modeled as a random oracle. In the security proof, U is simulated as a uniformly random value U, since the simulator, $\mathcal{S}$, does not know K_i or $K_i + R$. This means that if the environment later queries both K_i and $(K_i + R)$ to the random oracle then they could distinguish, as $\mathcal{S}$ would not be able to detect this, so the response would be inconsistent with the simulated U. However, with a **Key Query** command in the functionality, the simulator can detect this (based on the technique from [104]):

- For each query Q, $\mathcal{S}$ looks up all previous queries Q_i, and sends $(Q + Q_i)$ to the **Key Query** of the functionality.
- If **Key Query** is successful then $\mathcal{S}$ knows that $Q + Q_i = R$, so can program the response $\mathsf{H}(Q)$ such that $\mathsf{H}(Q) + \mathsf{H}(Q_i) = U$, as required.

4.7.2 Security

In this section we formalize the security of the implementation of the **Prep** command of $\mathcal{F}_{\text{n-TinyOT}}$ in our $\Pi_{\text{n-TinyOT}}$ protocol. More concretely, we focus on the consistency check in the production of m random bits. This guarantees that the MAC keys are consistent, after which the triple generation protocol can be proven secure similarly to [58]. The exact deviations that are possible by a corrupt P_j in the bit generation are:

1. Provide inconsistent inputs $\boldsymbol{r}^j$ when acting as sender in the **Initialize** command of the $\mathcal{F}_{\Delta\text{-ROT}}$ instances with two different honest parties.
2. Input inconsistent shares $b^j_\ell, \ell \in [m]$ or $r^j_h, h \in [\kappa]$ when acting as receiver in the **Extend** command of $\mathcal{F}_{\Delta\text{-ROT}}$ with two different honest parties.

Note that in both of these cases, we are only concerned when the other party in the $\mathcal{F}_{\Delta\text{-ROT}}$ execution is honest, as if both parties are corrupt then $\mathcal{F}_{\Delta\text{-ROT}}$ does not need to be simulated in the security proof. We should also remark that preventing the first attack in the production of the random bits extends to preventing it everywhere else in the protocol, as the $\boldsymbol{r}^j$ values are fixed in the **Initialize** phase.

These two attacks are modelled by defining $\boldsymbol{r}^{j,i}, b^{j,i}_\ell$ and $r^{j,i}_h$ to be the *actual* inputs used by a corrupt P_j in the above two cases. Without loss of generality, we pick a honest party P_{i_0} and

fix $b_\ell^j = b_\ell^{j,i_0}, r_h^j = r_h^{j,i}, \boldsymbol{r}^j = \boldsymbol{r}^{j,i_0}$ to be the inputs by P_j that should be consistent with every other honest party. Let I be the set of corrupted parties. For each $j \in I$, we can resume the previous statements by defining the values:

$$\begin{aligned}
&\Delta^{j,i_0} = 0, \quad \Delta^{j,i} = \boldsymbol{r}^{j,i} + \boldsymbol{r}^j, \quad i \notin (I \cup i_0) \\
&\delta_\ell^{j,i_0} = 0, \quad \delta_\ell^{j,i} = b_\ell^{j,i} + b_\ell^j, \quad \ell \in [m], i \notin (I \cup i_0) \\
&\hat{\delta}_\ell^{j,i_0} = 0, \quad \hat{\delta}_h^{j,i} = r_h^{j,i} + r_h^j, \quad h \in [\kappa], i \notin (I \cup i_0).
\end{aligned}$$

Note that $\Delta^{j,i}$ is fixed in the initialization of $\mathcal{F}_{\Delta\text{-ROT}}$, whilst $\delta_\ell^{j,i}$ may be different for every OT. Whenever P_i and P_j are both corrupt, or both honest, for convenience we define $\Delta^{j,i} = 0$ and $\delta^{j,i} = 0$. The above means that the outputs of $\mathcal{F}_{\Delta\text{-ROT}}$ with (P_i, P_j) then satisfy

$$M_i^j(b_\ell^{j,i}) = K_j^i(b_\ell^{j,i}) + b_\ell^{j,i} \cdot \boldsymbol{r}^{i,j},$$

or, equivalently,

$$M_i^j(b_\ell^j + \delta_\ell^{j,i}) = K_j^i(b_\ell^j + \delta_\ell^{j,i}) + (b_\ell^j + \delta_\ell^{j,i}) \cdot (\boldsymbol{r}^i + \Delta^{i,j}),$$

where $\delta^{j,i} \neq 0$ if P_j (the receiver) cheated, and $\Delta^{i,j} \neq 0$ if P_i (the sender) cheated. Recall from Section 4.4.1 that $M_i^j(b_\ell^j + \delta_\ell^{j,i})$ represents the receiver's MAC on the value $b_\ell^j + \delta_\ell^{j,i}$ and $K_j^i(b_\ell^j + \delta_\ell^{j,i})$ and represents the sender's MAC key on that same value.

We start by assuming that the corrupted party in the couple (P_i, P_j) running $\mathcal{F}_{\Delta\text{-ROT}}$ is the sender P_j, trying to have inconsistent correlations $R^{j,i}$ with different honest parties $P_i, i \notin I$. We prove the inconsistency impossible in the next claim:

Claim 4.7.1. *If the **Prep** step of $\Pi_{\text{n-TinyOT}}$ succeeds then all the global keys R^j are consistent and well-defined, i.e. $\Delta^{j,i} = 0$ for every $i, j \in [n]$.*

Proof. We enumerate the possible deviations by the Adversary affecting the check $\sum_{i=1}^n Z_j^i = 0$ in Step 3g with which we want to catch inconsistent $R^{j,i}$ values to different honest parties. These possible disruptions are the following two:

In Step 3e, the parties broadcast $\bar{C}^i$ values, every corrupted $P_\ell, \ell \in I$ can send instead some adversarial value $\hat{C}^\ell$ such that $\sum_{j=1}^n \hat{C}^j = C + e$, where e is some additive error of the Adversary's choice. Alternatively, a similar active deviation is to commit to $\hat{Z}_j^\ell$ values, $\ell \in I$, in such a way that $\sum_{\ell \in I} \hat{Z}_j^\ell = \sum_{\ell \in I} Z_j^\ell + E^j$.

An active P_j trying to cheat has to pass the aforementioned mentioned check, which becomes:

$$\begin{aligned}
0 = \sum_{i=1}^n \hat{Z}_j^i = E^j + Z_j^j + \sum_{i \neq j} Z_j^i = E^j + \left(\sum_{i \neq j} K_i^j(C^i) + (C + e + C^j) \cdot R^j \right) + \sum_{i \neq j} M_j^i(C^i) = \\
E^j + (C + e + C^j) \cdot R^j + \sum_{i \neq j} (K_i^j(C^i) + M_j^i(C^i)) = E^j + (C + e + C^j) \cdot R^j + \sum_{i \neq j} C^i \cdot R^{j,i} = \\
E^j + (C + e + C^j + \sum_{i \neq j} C^i) \cdot R^j + \sum_{i \neq j} C^i \cdot \Delta^{j,i} = E^j + e \cdot R^j + \sum_{i \neq j} C^i \cdot \Delta^{j,i}
\end{aligned}$$

As having inconsistent keys requires that there exists $i_0, i_1 \notin I$ such that $\Delta^{j,i_0} \neq \Delta^{j,i_1} \neq 0$, the attack would require the adversary to set $E^j + e \cdot R^j = C^{i_0} \cdot \Delta^{j,i_0} + C^{i_1} \cdot \Delta^{j,i_1}$. But this is negligible in κ, as the only information the adversary has about $C^{i_0}, C^{i_1} \in \mathbb{F}_{2^\kappa}$ at the time of committing to the values $\hat{Z}_j^\ell, \ell \in I$ is that they are two uniform additive shares of C, due to the rerandomization in Step 3d. ■

Finally, we prove that a corrupted receiver P_j cannot input inconsistent values $b_\ell^{j,i}$ to different honest parties.

Claim 4.7.2. *If the **Prep** step of $\Pi_{\text{n-TinyOT}}$ succeeds, every ordered pair (P_i, P_j) holds a secret sharing of $b_\ell^j \cdot R^i$ for every $\ell \in [m]$. In other words, $\delta_\ell^{j,i} = 0$ for every i, j, ℓ.*

Proof. For every ordered pair (P_i, P_j) we can define P_j's MAC on $[C^j]_{j,i}$ as

$$M_i^j(C^j) = \sum_{\ell=1}^{m} \chi_\ell \cdot M_i^j(b_\ell^{j,i}) + \sum_{h=1}^{\kappa} X^{h-1} \cdot M_i^j(r_h^{j,i})$$

and P_i's key on the same value as:

$$K_j^i(C^j) = \sum_{\ell=1}^{m} \chi_\ell \cdot K_j^i(b_\ell^{j,i}) + \sum_{h=1}^{\kappa} X^{h-1} \cdot K_j^i(r_h^{j,i})$$

In Step 3f of **Bits**, an adversarial P_j can also commit to incorrect MACs $\hat{Z}_i^j(c^j) = M_i^j(c^j) + E_i^j$ and $\hat{C}^j = C^j + e^j$. Nevertheless, in order to succeed an attack, the check $\hat{Z}_i^j = K_j^i(C^j) + \hat{C}^j \cdot R^i$ from Step 3g would have to hold. This check implies the following:

$$\begin{aligned}
& M_i^j(C^j) + E_i^j = K_j^i(C^j) + (C^j + e^j) \cdot R^i \\
\Leftrightarrow\ & E_i^j + (C^j + e^j) \cdot R^i = M_i^j(C^j) + K_j^i(C^j) = (\sum_{\ell=1}^{m} \chi_\ell \cdot b_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot r_h^{j,i}) \cdot R^i \\
& \Leftrightarrow E_i^j = \left(C^j + e^j + \sum_{\ell=1}^{m} \chi_\ell \cdot (b_\ell^j + \delta_\ell^{j,i}) + \sum_{h=1}^{\kappa} X^{h-1} \cdot (r_h^j + \hat{\delta}_h^{j,i}) \right) \cdot R^i \\
& \qquad = (e^j + \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i}) \cdot R^i
\end{aligned}$$

An active P_j has then just two options to cheat P_i, both with only probability $2^{-\kappa}$ to succeed:

1. Setting $E_i^j = (e^j + \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i}) \cdot R^i \neq 0$, which requires guessing the string $R^i \in \mathbb{F}_{2^\kappa}$ kept secret by the honest party P_i.

2. Setting $E_i^j = 0$ and $e_j = \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i}$ for every $i \notin I$. As $\delta_\ell^{j,i_0} = \hat{\delta}_h^{j,i_0} = 0$, this implies that $e_j = 0$. Thus, for every $i \notin (I \cup i_0)$ it needs to hold that

$$0 = \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i} = \sum_{h=1}^{\kappa} X^{h-1} \cdot (\hat{\delta}_h^{j,i} + \sum_{\ell=1}^{m} \delta_\ell^{j,i} \cdot \chi_{\ell,h}),$$

where the $\chi_{\ell,h}$ values are defined in such a way that $\chi_\ell = \sum_{h=1}^{\kappa} X^{h-1} \cdot \chi_{\ell,h}$. This would need that, for every $h \in [\kappa]$:

$$\hat{\delta}_h^{j,i} = \sum_{\ell=1}^{m} \delta_\ell^{j,i} \cdot \chi_{\ell,h} \in \mathbb{F}_2,$$

which can only happen with probability 1/2 for each of them, as $\chi_{\ell,h} \in \mathbb{F}_2$ are uniformly random sampled field elements after the deviations $\hat{\delta}_h^{j,i}, \delta_\ell^{j,i}$ have been defined.

■

4.7.3 Parameters

Based on the analysis from previous works [58, 59, 128], if roughly 1 million triples are created at once then the buckets in the cut-and-choose stages can be of size $B = 3$, to guarantee security except with probability 2^{-40}. The additional cut-and-choose parameter c can be as low as 3, so is insignificant as we initially need $m' = B^2 m + c$ triples to produce m final triples.

4.7.4 Communication Complexity

Here we analyse the communication complexity of $\Pi_{\text{n-TinyOT}}$. The cost of creating one shared random bit is the same as one invocation of the `extend` command in $\mathcal{F}_{\Delta\text{-ROT}}$ between all pairs of parties, giving $n(n-1)(\kappa+s)$ bits (we ignore the consistency check, since this cost amortizes away when creating many bits).

The cost of one triple (not counting the bucketing stage), is 3 calls to $\mathcal{F}_{\Delta\text{-ROT}}$ between every pair of parties for authenticating shares of (x, y, z), plus sending one correction bit between every pair of parties, giving $n(n-1)(3(\kappa+s)+1)$ bits. This is then multiplied by approximately B^2 to account for the bucketing. When creating a batch of at least a million triples (with $s = 40$), we can set $B = 3$, so the overall cost per party is around $(n-1)27 \cdot (\kappa+s)$ bits.

We remark that when checking a large number of MACs using Π_{Open} or Π_{Open}^i, the checks can be batched together, by first computing a random linear combination of all MACs, and checking the MAC on this, as in e.g. [44, 80]. This means that the cost of checking many MACs is roughly the cost of checking one, which is why we did not factor the MAC checks into the cost of the bucketing stage.

4.7.5 Round Complexity

Initializing the correlated OTs can be done with any 2-round OT protocol. Extending the correlated OTs using [107] and [11] takes 3 rounds. Note that when authenticating random bits, the s additional bits in the consistency check can be created in parallel with the original m bits, giving an overall cost of 5 rounds for random bits.

The triple generation consists of one set of correlated OTs (2 + 3 rounds), plus 1 round, plus another round of correlated OTs (3 rounds). Then there are 2 rounds for $\mathcal{F}_{\text{Rand}}$ in the bucketing

Protocol $\Pi_{\text{n-TinyOT Input}}$

1. Create a shared random bit $[b]$.
2. Open b to P_i using Π^i_{Open}.
3. P_i broadcasts $d = x - b$.
4. All parties compute $[x] = [b] + d$.

Figure 4.19: Input protocol for TinyOT-style Multi-Party Computation

(which can all be done in parallel), one round for the openings in step 2a and one round for step 2c. The openings in step 2a can be merged with the previous round. This gives a total of 13 rounds.

4.7.6 Realizing General Secure Computation

The previous protocol can easily be used to implement a general secure computation functionality such as $\mathscr{F}_{\text{Bit}\times\text{Bit}}$. The main feature missing is the ability for parties to provide inputs, since we only need to create random bits and triples for our application to garbled circuits. However, this is easy to do with a standard technique: if P_i wishes to secret-share an input x, the parties do as in $\Pi_{\text{n-TinyOT Input}}$, described in Figure 4.19.

CHAPTER 5

MULTI-PARTY COMPUTATION WITH SHORT KEYS

This chapter is based on joint work with Carmit Hazay, Emmanuela Orsini and Peter Scholl [69], which was presented at CRYPTO 2018.

In this chapter we investigate designing MPC protocols where *an arbitrary threshold t for the number of corrupted parties can be chosen*, which are practical both when n is very large, and also for small to medium sizes of n. Specifically, we ask the question:

> *Can we design* concretely efficient *MPC protocols where the performance improves gracefully as the number of honest parties increases?*

In the semi-honest case most protocols either require $t < n/2$, in which case unconditionally secure protocols [28, 38] based on Shamir secret-sharing can be used, or support any choice of t up to $n-1$, as in computationally secure protocols based on oblivious transfer [62, 64]. Interestingly, within these two ranges, the efficiency of most practical semi-honest protocols *does not depend on* t. For instance, the GMW [64] protocol (and its many variants) is *full-threshold*, so supports any $t < n$ corruptions. However, we *do not know* of any practical protocols with threshold, say, $t = \frac{2}{3}n$, or even $t = n/2 + 1$, that are more efficient than full-threshold GMW-style protocols.

One exception to this is when the number of parties becomes very large, in which case protocols based on *committees* can be used. In this approach, introduced by Bracha [33], first a random committee of size $n' \ll n$ is chosen. Then every party secret-shares its input to the parties in the committee, which runs a protocol secure against $t < n'$ corrupted parties to obtain the result. The committee size n' must be chosen to ensure (with high probability) that not the whole committee is corrupted, so clearly a lower threshold t allows for smaller committees, giving

significant efficiency savings. However, this technique is only really useful when n is very large, at least in the hundreds or thousands.

Note that the performance of an MPC protocol can be measured both in terms of *communication overhead* and *computational overhead*. Using fully homomorphic encryption [61], it is possible to achieve very low communication overhead that is independent of the circuit size [9] even in the malicious setting, but for reasonably complex functions FHE is impractical due to very high computational costs. On the other hand, practical MPC protocols typically communicate for every AND gate in the circuit, and use *oblivious transfer* (OT) to carry out the computation. Fast OT extension techniques allow a large number of secret-shared bit multiplications[1] to be performed using only symmetric primitives and an amortized communication complexity of $O(\kappa)$ [74] or $O(\kappa/\log\kappa)$ [51, 83] bits, where κ is a computational security parameter. This leads to an overall communication complexity which grows with $O(n^2\kappa/\log\kappa)$ bits per AND gate in protocols based on secret-sharing following the GMW [64] style, and $O(n^2\kappa)$ in those based on garbled circuits in the style of BMR [18, 25, 132].

5.1 Introduction

Our main idea towards achieving the above goal is to build a secure multi-party protocol with h honest parties, by distributing secret key material so that each party only holds a *small part of the key*. Instead of basing security on secret keys held by each party individually, we then base security on the *concatenation of all honest parties' keys*.

As a toy example, consider the following simple distributed encryption of a message m under n keys:

$$\mathsf{E}_k(m) = \bigoplus_{i=1}^{n} \mathsf{H}(i, k_i) \oplus m$$

where H is a suitable hash function and each key $k_i \in \{0,1\}^\ell$ belongs to party P_i. In the full-threshold setting with up to $n-1$ corruptions, to hide the message we need each party's key to be of length $\ell = 128$ to achieve 128-bit computational security. However, if only $t < n-1$ parties are corrupted, it seems that, intuitively, an adversary needs to guess all $h := n-t$ honest parties' keys to recover the message, and potentially each key k_i can be *much less than 128 bits* long when h is large enough. This is because the 'obvious' way to try to guess m would be to brute force all h keys until decrypting 'successfully'.

In fact, recovering m when there are h unknown keys corresponds to solving an instance of the *regular syndrome decoding problem* [12], which is related to the well-known *learning parity with noise* (LPN) problem, and believed to be hard for suitable choices of parameters.

[1] Note that OT is equivalent to secret-shared bit multiplication, and when constructing MPC it is more convenient to use the latter definition.

5.1.1 Our Contributions

In this work we use the above idea of short secret keys to design new MPC protocols in both the constant round and non-constant round settings, which improve in efficiency as the number of honest parties increases. Our contributions are captured by the following:

GMW-STYLE MPC WITH SHORT KEYS (SECTION 5.3). We present a GMW-style MPC protocol for binary circuits, where multiplications are done with OT extension using short symmetric keys. This reduces the communication complexity of OT extension-based GMW from $O(n^2\kappa/\log\kappa)$ [83] to $O(nt\ell)$, where the key length ℓ decreases as the number of honest parties, $h = n - t$, increases. When h is large enough, we can even have ℓ *as small as* 1.

To construct this protocol, we first analyse the security of the IKNP OT extension protocol [74] when using short keys, and formalise the leakage obtained by a corrupt receiver in this case. We then show how to use this version of 'leaky OT' to generate multiplication triples using a modified version of the GMW method, where pairs of parties use OT to multiply their shares of random values. We also optimize our protocol by reducing the number of communication channels using two different-sized committees, improving upon the standard approach of choosing one committee to do all the work.

MULTI-PARTY GARBLED CIRCUITS WITH SHORT KEYS (SECTION 5.4). Our second contribution is the design of a constant round, BMR-style (see Section 2.6) protocol based on garbled circuits with short keys. Our offline phase uses the multiplication protocol from the previous result in order to generate the garbled circuit, using secret-shared bit and bit/string multiplications as done in previous works [25, 70], with the exception that the keys are shorter. In the online phase, we then use the LPN-style assumption to show that the combination of all honest parties' ℓ-bit keys suffices to obtain a secure garbling protocol. This allows us to save on the key length as a function of the number of honest parties.

As well as reducing communication with a smaller garbled circuit, we also reduce computation when evaluating the circuit, since each garbled gate can be evaluated with only $O(n^2\ell/\kappa)$ block cipher calls (assuming the ideal cipher model), instead of $O(n^2)$ when using κ-bit keys. For this protocol, ℓ *can be as small as 5* when n is large enough, giving a significant saving over 128-bit keys used previously.

It is worth mentioning that the techniques presented here were later successfully extended to the malicious setting by the same authors [68]. We achieved so by efficiently constructing short message authentication codes and carefully using them to produce secret-shared bit multiplications.

5.1.1.1 Concrete Efficiency Improvements.

The efficiency of our protocols depends on the total number of parties, n, and the number of honest parties, h, so there is a large range of parameters to explore when comparing with other works. We discuss this in more detail in Section 5.5. Our protocols seem most significant in the *dishonest majority* setting, since when there is an honest majority there are unconditionally secure protocols with $O(n \log n)$ communication overhead and reasonable computational complexity e.g. [45], whilst our protocols have $\Omega(nt)$ communication overhead.

Our GMW-style protocol starts to improve upon previous protocols when we reach $n = 20$ parties and $t = 14$ corruptions: here, our triple generation method requires less than *half the communication cost* of the fastest GMW-style protocol based on OT extension [51] tolerating up to $n-1$ corruptions. When the number of honest parties is large enough, we can use *1-bit keys*, giving a *25-fold reduction* in communication over previous protocols when $n = 400$ and $t = 280$. In addition, we describe a simple threshold-t variant of GMW-style protocols, which our protocol still outperforms by 1.1x and 13x, respectively, in these two scenarios.

For our constant round protocol, with $n = 20, t = 10$ we can use 32-bit keys, so the size of each garbled AND gate is 1/4 the size of [25]. As n increases the improvements become greater, with a *16-fold reduction* in garbled AND gate size for $n = 400, t = 280$. We also reduce the communication cost of *creating* the garbled circuit. Here, the improvement starts at around 50 parties, and goes up to a 7 times reduction in communication when $n = 400, t = 280$. Note that our protocol does incur a slight additional overhead, since we need to use extra 'splitter gates' [126], but this cost is relatively small.

To demonstrate the practicality of our approach, we also present an implementation of the online evaluation phase of our constant-round protocol for key lengths ranging between $1-4$ bytes, and with an overall number of parties ranging from $15-1000$; more details can be found in Section 5.5.

5.1.1.2 Applications.

Our techniques seem most useful for large-scale MPC with around 70% corruptions, where we obtain the greatest concrete efficiency improvements. An important motivation for this setting is privacy-preserving statistical analysis of data collected from a large network with potentially thousands of nodes. In scenarios where the nodes are not always online and connected, our protocols can also be used with the 'random committee' approach discussed earlier, so only a small subset of, say, a hundred nodes need to be online and interacting during the protocol.

An interesting example is safely measuring the Tor network [53] which is amongst the most popular tools for digital privacy, consisting of more than 6000 relays that can opt-in for providing statistics about the use of the network. Nowadays and due to privacy risks, the statistics collected over Tor are generally poor: There is a reduced list of computed functions and only a minority of the relays provide data, which has to be obfuscated before publishing [53]. Hence, the

statistics provide an incomplete picture which is affected by a noise that scales with the number of relays. Running MPC in this setting would enable for more complex, accurate and private data processing, for example through anomaly detection and more sophisticated censorship detection. Moreover, our protocols are particularly well-suited to this setting since all relays in the network must be connected to one another already, by design.

Another possible application is for securely computing the interdomain routing within the Border Gateway Protocol (BGP), which is performed at a large scale of thousands of nodes. A recent solution in the dishonest majority setting [8] centralizes BGP so that two parties run this computation for all Autonomous Systems. Our techniques allow scaling to a large number of systems computing the interdomain routing themselves using MPC, hence further reducing the trust requirements.

5.1.1.3 Decisional Regular Syndrome Decoding problem.

The security of our protocols relies on the *Decisional Regular Syndrome Decoding (DRSD)* problem, which, given a random binary matrix $\boldsymbol{H}$, is to distinguish between the syndrome obtained by multiplying $\boldsymbol{H}$ with an error vector $\boldsymbol{e} = (\boldsymbol{e}_1 \| \cdots \| \boldsymbol{e}_h)$ where each $\boldsymbol{e}_i \in \{0,1\}^{2^\ell}$ has Hamming weight one, and the uniform distribution. This can equivalently be described as distinguishing $\bigoplus_{i=1}^{h} \mathsf{H}(i, k_i)$ from the uniform distribution, where H is a random function and each k_i is a random ℓ-bit key (as in the toy example described earlier).

We remark that when h is large enough, the problem is *unconditionally hard* even for $\ell = 1$, which means for certain parameter choices in our GMW-based protocol we can use 1-bit keys *without introducing any additional assumptions*. This introduces a significant saving in our triple generation protocol.

Overall, our approach demonstrates a new application of LPN-type assumptions to efficient MPC without introducing asymmetric operations. Our techniques may also be useful in other distributed applications where only a small fraction of nodes are honest.

5.1.1.4 Additional related work

Another work which applies a similar assumption to secure computation is that of Applebaum [3], who built garbled circuits with the free-XOR technique in the standard model under the LPN assumption. Conceptually, our work differs from Applebaum's since our focus is to improve the efficiency of multi-party protocols with fewer corruptions, whereas in [3], LPN is used in a more modular way in order to achieve encryption with stronger properties and under a more standard assumption.

In a recent work [106], Nielsen and Ranellucci designed a protocol in the dishonest majority setting with malicious, adaptive security in the presence of $t < cn$ corruption for $t \in [0,1)$. Their protocol is aimed to work with a large number of parties and uses committees to obtain a protocol

with poly-logarithmic overhead. This protocol introduces high constants and is not useful for practical applications.

Finally, Ben-Efraim and Omri [27] also explore how to optimize garbled circuits in the presence of non-full-threshold adversaries. By using deterministic committees they achieve AND gates of size $4(t+1)\kappa$, where κ is the computational security parameter. By using the same technique we achieve a size of $4(t+h)\ell$, where $\ell \ll \kappa$ depends on h, a parameter for the minimum number of honest parties in the committee. The rest of their results apply only to the honest majority setting.

5.1.2 Technical Overview

In what follows we introduce the technical side of our results in more detail.

5.1.2.1 Leaky oblivious transfer (OT).

We first present a two-party secret-shared bit multiplication protocol, based on a variant of the IKNP OT extension protocol [74] with short keys. Our protocol performs a batch of r multiplications at once. Namely, the parties create r correlated OTs on ℓ-bit strings using the OT extension technique of [74], by transposing a matrix of ℓ OTs on r-bit strings and swapping the roles of sender and receiver. In contrast to the IKNP OT extension and followups, that use κ 'base' OTs for computational security parameter κ, we use $\ell = O(\log\kappa)$ base OTs.

This protocol leaks some information on the global secret $\Delta \leftarrow \{0,1\}^\ell$ picked by the receiver, as well as the inputs of the receiver. Roughly speaking, the leakage is of the form $\mathsf{H}(i,\Delta) + x_i$, where $x_i \in \{0,1\}$ is an input of the receiver and H is a hash function with 1-bit output. Clearly, when ℓ is short this is not secure to use on its own, since all of the receiver's inputs only have ℓ bits of min-entropy (based on the choice of Δ).

5.1.2.2 MPC from leaky OT.

We then show how to apply this leaky two-party protocol to the multi-party setting, whilst preventing any leakage on the parties shares. The main observation is that, when using additive secret-sharing, we only need to ensure that the *sum* of all honest parties' shares is unpredictable; if the adversary learns just a few shares, they can easily be rerandomized by adding pseudorandom shares of zero, which can be done non-interactively using a PRF. However, we still have a problem, which is that in the standard GMW approach, each party P_i uses OT to multiply their share x^i with every other party P_j's share y^j. Now, there is leakage on the *same share* x^i from each of the OT instances between all other parties, which seems much harder to prevent than leakage from just a single OT instance.

To work around this problem, we have the parties add shares of zero to their x^i inputs *before* multiplying them. So, every pair (P_i, P_j) will use leaky OT to multiply $x^i \oplus s^{i,j}$ with y^j, where

$s^{i,j}$ is a random share of zero satisfying $\bigoplus_{i=1}^{n} s^{i,j} = 0$. This preserves correctness of the protocol, because the parties end up computing an additive sharing of:

$$\bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n}(x^i \oplus s^{i,j})y^j = \bigoplus_{j=1}^{n} y^j \bigoplus_{i=1}^{n}(x^i \oplus s^{i,j}) = xy.$$

This also effectively removes leakage on the individual shares, so we only need to be concerned with the *sum* of the leakage on all honest parties' shares: This turns out to be of the form $\bigoplus_{i=1}^{n}(\mathsf{H}(i,\Delta_i) + x^i)$, which is pseudorandom under the decisional regular syndrome decoding assumption.

We realize our protocol using a hash function with a polynomial-sized domain, so that is can be implemented using a CRS which simply outputs a random lookup-table. This means that, unlike when using the IKNP protocol, we do not need to rely on a random oracle or a correlation robustness assumption.

When the number of parties is large enough, we can improve our triple generation protocol using *random committees*. In this case the amortized communication cost is $\leq n_h n_1(\ell + \ell\kappa/r + 1)$ bits per multiplication where we need to choose two committees of sizes n_h and n_1 which have at least h and 1 honest parties, respectively.

5.1.2.3 Garbled circuits with short keys.

We next revisit the multi-party garbled circuits technique by Beaver, Micali and Rogaway, described in Section 2.6 and known as BMR, where essentially all the parties jointly garble using one set of keys each. This method was recently improved in a sequence of works [25, 70, 93, 96], within which [25, 70] further support the Free-XOR property.

Our garbling method uses an expansion function $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}^{n\ell+1}$, where ℓ is the length of each parties' keys used as wire labels in the garbled circuit. To garble a gate, the hash values of the input wire keys $k^i_{u,b}$ and $k^i_{v,b}$ are XORed over i and used to mask the output wire keys.

Specifically, for an AND gate g with input wires u, v and output wire w, the 4 garbled rows $\tilde{g}_{a,b}$, for each $(a,b) \in \{0,1\}^2$, are computed as

$$\tilde{g}_{a,b} = \left(\bigoplus_{i=1}^{n} \mathsf{H}(i,b,k^i_{u,a}) \oplus \mathsf{H}(i,a,k^i_{v,b})\right) \oplus (c, k^1_{w,c}, \ldots, k^n_{w,c}).$$

Security then relies on the DRSD assumption, which implies that the sum of h hash values on short keys is pseudorandom, which suffices to construct a secure garbling method with h honest parties.

Using this assumption instead of a PRF (as in recent works) comes with difficulties, as we can no longer garble gates with arbitrary fan-out, or use the Free-XOR technique, without degrading the DRSD parameters. To allow for arbitrary fan-out circuits with our protocol we use *splitter gates*, which take as input one wire w and provide two outputs wires u, v, representing the same

Functionality $\mathscr{F}^r_{\text{Zero}}(\mathscr{P})$

On receiving (zero) from all parties in $\mathscr{P} = \{P_1, \ldots, P_n\}$:

1. Sample random shares $\boldsymbol{s}^2, \ldots, \boldsymbol{s}^n \leftarrow \{0,1\}^r$ and let $\boldsymbol{s}^1 = \boldsymbol{s}^2 \oplus \cdots \oplus \boldsymbol{s}^n$
2. Send $\boldsymbol{s}^i$ to party P_i

Figure 5.1: Random zero sharing functionality.

wire value. Splitter gates were previously introduced in [126] as a fix for an error in the original BMR paper. We stress that transforming a general circuit description into a circuit with only fan-out-1 gates requires adding at most a single splitter gate per AND or XOR gate.

The restriction to fan-out-1 gates and the use of splitter gates additionally allows us to garble XOR gates for free in BMR without relying on circular security assumptions or correlation-robust hash functions, based on the FlexOR technique [84] where each XOR gate uses a unique offset. Furthermore, the overhead of splitter gates is very low, since garbling a splitter gate does not use the underlying MPC protocol: shares of the garbled gate can be generated non-interactively. We note that this observation also applies to Yao's garbled circuits, but the overhead of adding splitter gates there is more significant; this is because in most 2-party protocols, the *size* of the garbled circuit is the dominant cost factor, whereas in multi-party protocols the main cost is *creating* the garbled circuit in a distributed manner.

5.2 Preliminaries

5.2.1 Security and Communication Models

We prove security of our protocols in the universal composability (UC) framework [35]. See Section 2.4 for a summary of this. We assume all parties are connected via secure, authenticated point-to-point channels, which is the default method of communication in our protocols. The adversary model we consider is a static, honest-but-curious adversary who corrupts a subset $A \subset [n]$ of parties at the beginning of the protocol. We denote by $\bar{A}$ the subset of honest parties, and define $h = |\bar{A}| = n - t$.

5.2.2 Random Zero-Sharing

Our protocols require the parties to generate random additive sharings of zero, as in the $\mathscr{F}_{\text{Zero}}$ functionality in Figure 5.1. This can be done efficiently using a PRF F, with interaction *only* during a setup phase, as in [6]. We do this by asking each party P_i to send a random PRF key $k_{i,j}$ to every other party P_j. Next, P_i defines its share by $\bigoplus_{j \neq i}(F_{k_{i,j}}(\tau) \oplus F_{k_{j,i}}(\tau))$ where τ is an index that identifies the generated share. It is simple to verify that all the shares XOR to zero since each PRF value is used exactly twice. Moreover, privacy holds in the presence of any subset of $n-2$ corrupted parties because the respective values $F_{k_{l,l'}}$ and $F_{k_{l',l}}$ of honest parties P_l and

$P_{l'}$ are pseudorandom, which implies that their zero shares are also pseudorandom. Finally, the communication complexity of the setup phase amounts to sending $O(n^2)$ PRF keys, whilst creating the shares requires $2(n-1)$ PRF evaluations to produce κ bits.

5.2.3 Syndrome Decoding and Learning Parity with Noise

For completeness, we describe the Learning Parity with Noise (LPN) and Syndrome Decoding (SD) search problems, as well as how they relate to each other.

Definition 5.1 (Learning Parity with Noise (LPN))**.** Let $\ell, q \in \mathbb{N}$ and let χ be a distribution defined over $\mathbb{F}_2^q$. Sample $\boldsymbol{A} \leftarrow \mathbb{F}_2^{q\times\ell}$ uniformly random and $\boldsymbol{e} \in \mathbb{F}_2^q$ according to χ. Given $(\boldsymbol{A}, \boldsymbol{A}\cdot\boldsymbol{s}+\boldsymbol{e})$, the $\mathsf{LPN}_{q,\ell,\chi}$ problem is to recover $\boldsymbol{s} \in \mathbb{F}_2^\ell$ with noticeable probability.

Most commonly, the LPN problem is presented for χ consisting of q independent, identically distributed (i.i.d.) Bernoulli distributions with parameter $\tau \in (0, 0.5)$. In other words, if $\boldsymbol{e} = (e_1, \ldots, e_q)$, then each e_i is set to 1 with probability τ. Unlike most assumptions used in cryptography, the LPN problem is not known to be broken in the presence of a quantum adversary. As we will show below, the search LPN problem can also be presented as the problem of decoding random linear codes.

A binary $[m,k,d]_2$ linear code C is a k-dimensional subspace of $\mathbb{F}_2^m$, where m is the length of the code, k is its dimension as a vector subspace and d is its distance, i.e. the minimal non-zero Hamming distance between any two elements of C. We say $\boldsymbol{G} \in \mathbb{F}_2^{k\times m}$ is a generator matrix of a linear code C if $\boldsymbol{G}$'s rows are a basis of C. Given any *message* $\boldsymbol{s} \in \mathbb{F}_2^m$, we can *encode* it using C by computing $\boldsymbol{G}\cdot\boldsymbol{s} = \boldsymbol{x}$. We call any such $\boldsymbol{x} \in C$ a *codeword*.

Equivalently, C can be defined as the kernel of a full-rank matrix $\boldsymbol{H} \in \mathbb{F}_2^{(m-k)\times m}$, called a parity-check matrix of C. Given a vector $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{e} \in \mathbb{F}_2^m$, where $\boldsymbol{x} \in C$ is a codeword and $\boldsymbol{e}$ an error vector, the *syndrome* corresponding to $\boldsymbol{z}$ is the vector $\boldsymbol{y} = \boldsymbol{H}\cdot\boldsymbol{z} = \boldsymbol{H}\cdot\boldsymbol{x} + \boldsymbol{H}\cdot\boldsymbol{e} = \boldsymbol{H}\cdot\boldsymbol{e} \in \mathbb{F}_2^{m-k}$. Hence, the syndrome does not depend on the codeword, but only on the error vector. When the Hamming weight $\mathrm{wt}(\boldsymbol{e})$ of $\boldsymbol{e}$ is smaller than the error correction capability of C, that is $\mathrm{wt}(\boldsymbol{e}) \le \lfloor \frac{d-1}{2} \rfloor$, $\boldsymbol{y}$ is called a *correctable syndrome* and $\boldsymbol{z}$ can be uniquely decoded to $\boldsymbol{x}$. More formally, we can define a mapping

$$\begin{aligned} \mathsf{Syn} : \mathbb{F}_2^m &\longrightarrow \mathbb{F}_2^r \qquad\qquad (r = m-k) \\ \boldsymbol{e} &\longmapsto \boldsymbol{H}\cdot\boldsymbol{e} (= \boldsymbol{y}). \end{aligned}$$

When the domain of Syn is restricted to vectors of upper bounded Hamming weight, inverting Syn is strictly related to the problem of decoding the $[m,k,d]_2$ linear code with parity-check matrix $\boldsymbol{H}$, and this problem is equivalent to the average-case hardness of the following computational search problem.

Definition 5.2 (Syndrome Decoding (SD))**.** Let $r, h, m \in \mathbb{N}$. Sample a parity check matrix $\boldsymbol{H} \leftarrow \mathbb{F}_2^{r \times m}$ and $\boldsymbol{e} \leftarrow \mathbb{F}_2^m$ such that $\mathsf{wt}(\boldsymbol{e}) = h$. Given $(\boldsymbol{H}, \boldsymbol{H} \cdot \boldsymbol{e})$, the $\mathsf{SD}_{r,m,h}$ problem is to recover $\boldsymbol{e}$ with noticeable probability.

Let $\boldsymbol{G}$ be C's generator matrix and $\boldsymbol{s}$ a message prior to its encoding. As described above, the $\mathsf{SD}_{r,m,h}$ problem is then equivalent to recovering the message $\boldsymbol{s}$ from the noisy codeword $\boldsymbol{z} = \boldsymbol{G}\boldsymbol{s} + \boldsymbol{e}$. If we define $\tilde{\chi}$ to be the uniformly random distribution over the elements of $\mathbb{F}_2^m$ with Hamming weight h, then this is exactly $\mathsf{LPN}_{m-r,m,\tilde{\chi}}$. Actually, it turns out that defining LPN with such distribution is at least as hard as its more common presentation, where χ is made out of i.i.d. Bernoulli distributions [54, Lemma 3.6.]. Intuitively, this is due to the fact that if $\boldsymbol{e}$ is sampled according to i.i.d. Bernoulli distributions, then with noticeable probability $\boldsymbol{e}$ will also be a sample of $\tilde{\chi}$.

The extension of LPN from the binary field $\mathbb{F}_2$ to a prime field $\mathbb{F}_p$ is known as Learning with Errors [116], which has become a major assumption in lattice-based cryptography for realizing complex primitives such as homomorphic encryption. In this chapter, though, we are interested in a variant of the SD problem in which some further structure is added to the error vector $\boldsymbol{e}$.

5.2.4 Regular Syndrome Decoding Problem

We introduce the Regular Syndrome Decoding (RSD) problem and some of its properties.

Definition 5.3. A vector $\boldsymbol{e} \in \mathbb{F}_2^m$ is (m,h)-*regular* if $\boldsymbol{e} = (\boldsymbol{e}_1 \| \cdots \| \boldsymbol{e}_h)$ where each $\boldsymbol{e}_i \in \{0,1\}^{m/h}$ has Hamming weight one. We denote by $R_{m,h}$ the set of all the (m,h)-regular vectors in $\mathbb{F}_2^m$.

Definition 5.4 (Regular Syndrome Decoding (RSD))**.** Let $r, h, \ell \in \mathbb{N}$ with $m = h \cdot 2^\ell$, $\boldsymbol{H} \leftarrow \mathbb{F}_2^{r \times m}$ and $\boldsymbol{e} \leftarrow R_{m,h}$. Given $(\boldsymbol{H}, \boldsymbol{H}\boldsymbol{e})$, the $\mathsf{RSD}_{r,h,\ell}$ problem is to recover $\boldsymbol{e}$ with noticeable probability.

The decisional version of the problem, given below, is to distinguish the syndrome $\boldsymbol{H}\boldsymbol{e}$ from uniform.

Definition 5.5 (Decisional Regular Syndrome Decoding (DRSD))**.** Let $\boldsymbol{H} \leftarrow \mathbb{F}_2^{r \times m}$ and $\boldsymbol{e} \leftarrow R_{m,h}$, and let U_r be the uniform distribution on r bits. The $\mathsf{DRSD}_{r,h,\ell}$ problem is to distinguish between $(\boldsymbol{H}, \boldsymbol{H}\boldsymbol{e})$ and $(\boldsymbol{H}, U_r)$ with noticeable advantage.

5.2.4.1 Hash function formulation.

The DRSD problem can be equivalently described as distinguishing from uniform $\bigoplus_{i=1}^h \mathsf{H}(i, k_i)$ where $\mathsf{H} : [h] \times \{0,1\}^\ell \to \{0,1\}^r$ is a random hash function, and each $k_i \leftarrow \{0,1\}^\ell$. With this formulation, it is easier to see how the DRSD problem arises when using our protocols with short keys, since this appears when summing up a hash function applied to h honest parties' secret keys.

To see the equivalence, we can define a matrix $\boldsymbol{H} \in \mathbb{F}_2^{r \times h \cdot 2^\ell}$, where for each $i \in \{0, \ldots, h-1\}$ and $k \in [2^\ell]$, column $i \cdot 2^\ell + k$ of $\boldsymbol{H}$ contains $\mathsf{H}(i,k)$. Then, multiplying $\boldsymbol{H}$ with a random (m,h)-regular vector $\boldsymbol{e}$ is equivalent to taking the sum of H over h random inputs, as above.

5.2.4.2 Statistical hardness of DRSD.

We next observe that for certain parameters where the output size of $\boldsymbol{H}$ is sufficiently smaller than the min-entropy of the error vector $\boldsymbol{e}$, the distribution in the decisional problem is *statistically close to uniform*.

Lemma 5.2.1. *If $\ell = 1$ and $h \geq r + s$ then $\mathsf{DRSD}_{r,h,\ell}$ is statistically hard, with distinguishing probability 2^{-s}.*

Proof. Suppose $\ell = 1$ and $h \geq r + s$, so $m = 2h$. For a vector $\boldsymbol{e} = (\boldsymbol{e}_1 \| \cdots \| \boldsymbol{e}_h) \in R_{m,h}$, we can write each of the weight-1 vectors $\boldsymbol{e}_i \in \{0,1\}^2$ as $(e'_i, 1 - e'_i)$. An RSD sample $\boldsymbol{H}, \boldsymbol{y} = \boldsymbol{H}\boldsymbol{e}$ therefore defines a system of r linear equations in the h variables $\{e'_i\}_i$, and it can be shown that this simplifies to the form $\boldsymbol{y} = \boldsymbol{H}'\boldsymbol{e}' + \boldsymbol{c}$, where $\boldsymbol{e}' = (e'_1, \ldots, e'_h)$, by defining the j-th column of $\boldsymbol{H}' \in \mathbb{F}_2^{r \times h}$ to be the sum of columns $2j-1$ and $2j$ from $\boldsymbol{H}$, and $\boldsymbol{c}$ to be the sum of all even-indexed columns in $\boldsymbol{H}$. Note that $\boldsymbol{H}'$ is uniformly random because $\boldsymbol{H}$ is, and it is easy to show (e.g. [110, Lemma 1]) that the probability that $\boldsymbol{H}' \leftarrow \mathbb{F}_2^{r \times h}$ is not full rank is no more than 2^{-s} when $h \geq r + s$. Assuming that $\boldsymbol{H}'$ has full rank and $h \geq r$, $\boldsymbol{y} = \boldsymbol{H}'\boldsymbol{e}' + \boldsymbol{c}$ must be uniformly random because $\boldsymbol{e}'$ is. ■

For the general case of ℓ-bit keys, we use the following form of the leftover hash lemma.

Lemma 5.2.2 (Leftover Hash Lemma [72])**.** *Let $\boldsymbol{H} \leftarrow \mathbb{F}_2^{r \times m}$ and $\boldsymbol{e} \leftarrow \chi$, where χ is a distribution over $\mathbb{F}_2^m$ with min-entropy at least k. If $r \leq k - 2s$ then*

$$\Delta_{SD}((\boldsymbol{H}, \boldsymbol{H}\boldsymbol{e}), (\boldsymbol{H}, \boldsymbol{u})) \leq 2^{-s}$$

where $\boldsymbol{u} \leftarrow \mathbb{F}_2^r$ and Δ_{SD} is the statistical distance.

Note that if $\boldsymbol{e} \leftarrow R_{m,h}$ then we have $H_\infty(\boldsymbol{e}) = h\ell$. Applying Lemma 5.2.2 with $k = h\ell$, we obtain the following.

Corollary 5.2.1. *If $h \geq (r + 2s)/\ell$ then $\mathsf{DRSD}_{r,h,\ell}$ is statistically hard, with distinguishing probability 2^{-s}.*

5.2.4.3 Search-to-decision reduction.

For *all* parameter choices of DRSD, there is a simple reduction to the search version of the regular syndrome decoding problem with the same parameters.

Lemma 5.2.3. *Any efficient distinguisher for the $\mathsf{DRSD}_{r,h,\ell}$ problem can be used to efficiently solve $\mathsf{RSD}_{r,h,\ell}$.*

The proof (inspired by a similar result for LPN [4]) is a simplified version of previous reductions for syndrome decoding. We first recall the Goldreich-Levin hardcore-bit theorem.

Theorem 5.2.2 ([63])**.** *Let f be a one-way function. Then, given $(r, f(x))$ for uniformly random r and x, the inner product $\langle x, r\rangle$ over $\mathbb{F}_2$ is unpredictable.*

Proof. (of Lemma 5.2.3) Suppose $\mathscr{A}$ distinguishes between $(\boldsymbol{H}, \boldsymbol{He})$ and $(\boldsymbol{H}, U_r)$ with noticeable advantage δ. We construct an adversary $\mathscr{A}'$ that breaks the Goldreich-Levin hardcore bit of $f(\boldsymbol{e}) = (\boldsymbol{H}, \boldsymbol{He})$ by guessing the inner product $\langle \boldsymbol{e}, \boldsymbol{s}\rangle$ for some vector $\boldsymbol{s} \in \mathbb{F}_2^m$. On input $(\boldsymbol{H}, \boldsymbol{y} = \boldsymbol{He}, \boldsymbol{s})$, algorithm $\mathscr{A}'$ proceeds as follows:

1. Sample $\boldsymbol{t} \leftarrow \{0,1\}^r$
2. Compute $\boldsymbol{H}' = \boldsymbol{H} - \boldsymbol{t} \cdot \boldsymbol{s}^\top$
3. Run $\mathscr{A}$ on input $(\boldsymbol{H}', \boldsymbol{y})$
4. Output the same as $\mathscr{A}$

First notice that because $\boldsymbol{H}$ is uniformly random, $\boldsymbol{H}'$ is also. Secondly, $\boldsymbol{y} = \boldsymbol{He} = (\boldsymbol{H}' + \boldsymbol{t} \cdot \boldsymbol{s}^\top)\boldsymbol{e} = \boldsymbol{H}'\boldsymbol{e} + \boldsymbol{t} \cdot \langle \boldsymbol{s}, \boldsymbol{e}\rangle$. So, if $\langle \boldsymbol{s}, \boldsymbol{e}\rangle = 0$ then the input to $\mathscr{A}$ is a correct sample $(\boldsymbol{H}', \boldsymbol{H}'\boldsymbol{e})$, whereas if $\langle \boldsymbol{s}, \boldsymbol{e}\rangle = 1$ then the input is uniformly random. Therefore, it holds that:

$$\begin{aligned}
\Pr[\mathscr{A}'(\boldsymbol{H}, \boldsymbol{He}, \boldsymbol{r}) = \langle \boldsymbol{e}, \boldsymbol{r}\rangle] &= \Pr[\mathscr{A}'(\boldsymbol{H}, \boldsymbol{He}, \boldsymbol{r}) = 0 | \langle \boldsymbol{e}, \boldsymbol{r}\rangle = 0] \cdot \Pr[\langle \boldsymbol{e}, \boldsymbol{r}\rangle = 0] + \\
&\quad + \Pr[\mathscr{A}'(\boldsymbol{H}, \boldsymbol{He}, \boldsymbol{r}) = 1 | \langle \boldsymbol{e}, \boldsymbol{r}\rangle = 1] \cdot \Pr[\langle \boldsymbol{e}, \boldsymbol{r}\rangle = 1] \\
&= \frac{1}{2} \cdot (\Pr[\mathscr{A}(\boldsymbol{H}', \boldsymbol{H}'\boldsymbol{e}) = 0] + (1 - \Pr[\mathscr{A}(\boldsymbol{H}', U_r) = 0])) \\
&\geq \frac{1}{2} + \frac{\delta}{2}.
\end{aligned}$$

■

5.2.4.4 Multi-Secret RSD.

We now consider a variant of DRSD with multiple sets of secrets, where the matrix $\boldsymbol{H}$ is fixed for each sample. We then reduce this to the standard DRSD problem with the same parameters, with a security loss of the number of secrets.

Definition 5.6 (Multi-Secret DRSD)**.** Let $\boldsymbol{H} \leftarrow \mathbb{F}_2^{r \times m}$ and $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_q \leftarrow R_{m,h}$ (as in Definition 5.4). The q-$\mathsf{DRSD}_{r,h,\ell}$ problem is to distinguish between a tuple $(\boldsymbol{H}, \boldsymbol{He}_1, \ldots, \boldsymbol{He}_q)$ and $(\boldsymbol{H}, U_r^q)$ with noticeable advantage.

Lemma 5.2.4. *q-$\mathsf{DRSD}_{r,h,\ell}$ is reducible to $\mathsf{DRSD}_{r,h,\ell}$, where the reduction loses a tightness factor of q.*

Proof. The proof is based on a standard hybrid argument with a sequence of $q+1$ hybrid distributions, where each pair of neighbouring hybrids is indistinguishable based on DRSD.

The first hybrid, H_0, outputs $(\boldsymbol{H}, u_1, \ldots, u_q)$, where $\boldsymbol{H} \leftarrow \mathbb{F}_2^{r \times m}$ and $u_i \leftarrow \{0,1\}^r$, which is exactly the uniform distribution used in q-DRSD. In hybrid H_i, for $i = 1, \ldots, q$, we sample regular secrets $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_i$ and output $(\boldsymbol{H}, \boldsymbol{H}\boldsymbol{e}_1, \ldots, \boldsymbol{H}\boldsymbol{e}_i, u_{i+1}, \ldots, u_q)$. Note that H_q is the same as the real distribution in the q-DRSD problem. Any adversary $\mathscr{A}$ who distinguishes between H_i and H_{i+1} can be used to break $\mathsf{DRSD}_{r,h,\ell}$, as follows. The distinguisher $\mathscr{D}$ receives a DRSD challenge $(\boldsymbol{H}, \boldsymbol{y})$, then samples $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_i$ from the error distribution and random strings $u_{i+2}, \ldots, u_q \leftarrow \{0,1\}^r$. It then outputs $\mathscr{A}(\boldsymbol{H}, \boldsymbol{H}\boldsymbol{e}_1, \ldots, \boldsymbol{H}\boldsymbol{e}_i, \boldsymbol{y}, u_{i+2}, \ldots, u_q)$. The advantage of $\mathscr{D}$ against the DRSD problem is identical to that of $\mathscr{A}$. A standard argument then implies that any adversary who distinguishes H_0 and H_q with advantage δ can solve $\mathsf{DRSD}_{r,h,\ell}$ with advantage at least δ/q. ■

5.2.4.5 Extended Double-Key RSD.

In our final variant of RSD – used in the security proof of our BMR-style online phase – we consider multiple sets of secrets, and also give the adversary two challenges for each secret which captures the double use of each key in the garbling procedure. This means we cannot preserve the RSD parameters, and must reduce to $2\text{-}\mathsf{DRSD}_{2r,h,\ell}$. We also make a conceptual change, and specify the problem using a random hash function H with small domain (which can be modelled as a random oracle, or a random lookup table generated in a trusted setup phase which can be modelled as a common random string) instead of matrices and vectors. We switch to this notation in order to capture the computation made by the honest parties when garbling a gate.

Definition 5.7 (Extended Double-Key DRSD)**.** The extended double-key DRSD problem states that, for every fixed subset $S \subset [n]$ of size h, it holds that

$$\left(\mathsf{H}, \bigoplus_{i \in S} \mathsf{H}(i,0,k_i), \bigoplus_{i \in S} \mathsf{H}(i,0,k'_i), \bigoplus_{i \in S} \mathsf{H}(i,1,k_i), \bigoplus_{i \in S} \mathsf{H}(i,1,k'_i)\right) \quad \overset{\mathrm{c}}{\approx} \quad (\mathsf{H}, U_{4r}),$$

where $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}^r$ is a randomly sampled function, and $k_i, k'_i \leftarrow \{0,1\}^\ell$ for $i \in S$.

Lemma 5.2.5. *The extended double-key* DRSD *problem with parameters* (r,h,ℓ) *is reducible to* $2\text{-}\mathsf{DRSD}(2r,h,\ell)$.

Proof. Suppose there exists a set $S \subset [n]$ for which an adversary $\mathscr{A}$ distinguishes the above two distributions with noticeable advantage. We use $\mathscr{A}$ to construct a distinguisher $\mathscr{D}$ for the $2\text{-}\mathsf{DRSD}(2r,h,\ell)$ problem. $\mathscr{D}$ receives a challenge $(\boldsymbol{H}, \boldsymbol{y}_0, \boldsymbol{y}_1)$, where $\boldsymbol{H} \in \mathbb{F}_2^{2r \times m}$, $m = h \cdot 2^\ell$ and $\boldsymbol{y}_0, \boldsymbol{y}_1 \in \mathbb{F}_2^{2r}$. Write $\boldsymbol{H} = \begin{pmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{pmatrix}$ and $\boldsymbol{y}_j = \begin{pmatrix} \boldsymbol{z}_j \\ \boldsymbol{z}'_j \end{pmatrix}$. Define the hash function $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}^r$ so that $\mathsf{H}(s_i, b, k)$ is equal to column $2^\ell i + k$ (viewing k also as an integer in $[2^\ell]$) of the matrix $\boldsymbol{H}_b$, for each $s_i \in S$ and $b \in \{0,1\}$. For $i \in [n] \setminus S$, let the output of $\mathsf{H}(i,\cdot,\cdot)$ be uniformly random. The distinguisher then runs $\mathscr{A}$ with input

$$\left(\mathsf{H}, \boldsymbol{z}_0, \boldsymbol{z}'_0, \boldsymbol{z}_1, \boldsymbol{z}'_1\right),$$

Functionality $\mathscr{F}^{r,\ell}_{\Delta\text{-ROT}}$

After receiving $\Delta \in \{0,1\}^\ell$ from P_S and $(x_1,\ldots,x_r) \in \{0,1\}^r$ from P_R, do the following:

1. Sample $\boldsymbol{q}_i \leftarrow \{0,1\}^\ell$, for $i \in [r]$, and let $\boldsymbol{t}_i = \boldsymbol{q}_i \oplus x_i \cdot \Delta$.
2. Output $\boldsymbol{q}_i$ to P_S and $\boldsymbol{t}_i$ to P_R, for $i \in [r]$.

Figure 5.2: Functionality for oblivious transfer on random, correlated strings.

and outputs the same as $\mathscr{A}$. Notice that if the DRSD challenge is random then the input to $\mathscr{A}$ is random, whereas if the challenge is computed as $\boldsymbol{y}_j = \boldsymbol{H}\boldsymbol{e}_j$ for some regular error $\boldsymbol{e}_j$ and $j \in \{0,1\}$, then we have $\boldsymbol{z}_j = \boldsymbol{H}_0\boldsymbol{e}_j$ and $\boldsymbol{z}'_j = \boldsymbol{H}_1\boldsymbol{e}_j$, and by the definition of H, these values are equal to the sum of hash function outputs under some secret keys corresponding to $\boldsymbol{e}_j$. It follows that the distinguishing advantage of $\mathscr{D}$ is the same as that of $\mathscr{A}$. ■

5.3 GMW-Style MPC with Short Keys

In this section we design a protocol for generating multiplication triples over $\mathbb{F}_2$ using short secret keys, with reduced communication complexity as the number of honest parties increases. More concretely, we first design a leaky protocol for secret-shared two-party bit multiplication, based on correlated OT and OT extension techniques with short keys. This protocol is not fully secure and we precisely define the leakage obtained by the receiver. We next show how to use the leaky protocol to produce multiplication triples, removing the leakage by rerandomizing the parties' shares with shares of zero, and using the DRSD assumption. Finally, this protocol can be used with Beaver's multiplication triple technique [16] to obtain MPC for binary circuits with an amortized communication complexity of $O(nt\ell)$ bits per triple, where t is the threshold and ℓ is the secret key length. When the number of honest parties is large enough we can even use $\ell = 1$ and avoid relying on DRSD.

5.3.1 Leaky Two-Party Secret-Shared Multiplication

We first present our protocol for two-party secret-shared bit multiplication, based on a variant of the [74] OT extension protocol, modified to use short keys. With short keys we cannot hope for computational security based on standard symmetric primitives, because an adversary can search every possible key in polynomial time. Our goal, therefore, is to define the precise *leakage* that occurs when using short keys, in order to remove this leakage at a later stage.

5.3.1.1 OT extension and correlated OT

Recall that the main observation of the IKNP protocol for extending oblivious transfer [74] is that *correlated OT is symmetric*, so that κ correlated OTs on r-bit strings can be locally converted

into r correlated OTs on κ-bit strings. Secondly, a κ-bit correlated OT can be used to obtain an OT on chosen strings with computational security. The first stage of this process is abstracted away by the functionality $\mathscr{F}_{\Delta\text{-ROT}}$ in Figure 5.2.

Using IKNP to multiply an input bit x_k from the sender, P_A, with an input bit y_k from P_B, the receiver, P_B sends y_k as its choice bit to $\mathscr{F}_{\Delta\text{-ROT}}$ and learns $\boldsymbol{t}_k = \boldsymbol{q}_k \oplus y_k \cdot \Delta$. The sender P_A obtains $\boldsymbol{q}_k$, and then sends

$$d_k = \mathsf{H}(\boldsymbol{q}_k) \oplus \mathsf{H}(\boldsymbol{q}_k \oplus \Delta) \oplus x_k,$$

where H is a 1-bit output hash function. This allows the parties to compute an additive sharing of $x_k \cdot y_k$ as follows: P_A defines the share $\mathsf{H}(\boldsymbol{q}_k)$, and P_B computes $\mathsf{H}(\boldsymbol{t}_k) \oplus y_k \cdot d_k$. This can be repeated many times with the same Δ to perform a large batch of $\mathsf{poly}(\kappa)$ secret-shared multiplications, because the randomness in Δ serves to computationally mask each x with the hash values (under a suitable correlation robustness assumption for H). The downside of this is that for $\Delta \in \{0,1\}^\kappa$, the communication cost is $O(\kappa)$ bits per two-party bit multiplication, to perform the correlated OTs.

5.3.1.2 Variant with short keys

We adapt this protocol to use short keys by performing the correlated OTs on ℓ-bit strings, instead of κ-bit, for some small key length $\ell = O(\log \kappa)$ (we could have ℓ as small as 1). This allows $\mathscr{F}_{\Delta\text{-ROT}}$ to be implemented with only $O(\ell)$ bits of communication per OT instead of $O(\kappa)$.

Our protocol, shown in Figure 5.4, performs a batch of r multiplications at once. First the parties create r correlated OTs on ℓ-bit strings using $\mathscr{F}_{\Delta\text{-ROT}}$. Next, the parties hash the output strings of the correlated OTs, and P_A sends over the correction values d_k, which are used by P_B to convert the random OTs into a secret-shared bit multiplication. Finally, we require the parties to add a random value (from $\mathscr{F}_{\text{Zero}}$, shown in Figure 5.1) to their outputs, which ensures that they have a uniform distribution.

Note that if $\ell \in O(\log \kappa)$ then the hash function H_{AB} has a polynomial-sized domain, so can be described as a lookup table provided as a common input to the protocol by both parties. At this stage we do not make any assumptions about H_{AB}; this means that the leakage in the protocol will depend on the hash function, so its description is also passed to the functionality $\mathscr{F}_{\text{Leaky-2-Mult}}$ (Figure 5.3). We require H_{AB} to take as additional input an index $k \in [r]$ and a bit in $\{0,1\}$, to provide independence between different uses, and our later protocols require the function to be different in protocol instances between different pairs of parties (we use the notation H_{AB} to emphasize this).

5.3.1.3 Leakage

We now analyse the exact security of the protocol in Figure 5.4 when using short keys, and explain how this is specified in the functionality $\mathscr{F}_{\text{Leaky-2-Mult}}$ (Figure 5.3). Since a random share

Functionality $\mathscr{F}^{r,\ell}_{\text{Leaky-2-Mult}}$

INPUT: $(x_1,\ldots,x_r) \in \mathbb{F}_2^r$ from P_A and $(y_1,\ldots,y_r) \in \mathbb{F}_2^r$ from P_B.
COMMON INPUT: A hash function $\mathsf{H}_{AB} : [r] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}$.

1. Sample $\boldsymbol{z}^A, \boldsymbol{z}^B \leftarrow \mathbb{F}_2^r$ such that $\boldsymbol{z}^A + \boldsymbol{z}^B = \boldsymbol{x} * \boldsymbol{y}$ (where $*$ denotes component-wise product).
2. Output $\boldsymbol{z}^A$ to P_A and $\boldsymbol{z}^B$ to P_B.

Leakage: If P_B is corrupt:

1. Let $\boldsymbol{H} \in \mathbb{F}_2^{r \times 2^\ell}$ be defined so that entry (k,k') of $\boldsymbol{H}$ is $\mathsf{H}_{AB}(k, 1 \oplus y_k, \boldsymbol{t}_k \oplus k')$, where $\boldsymbol{t}_k \leftarrow \{0,1\}^\ell$.
2. Sample a random unit vector $\boldsymbol{e} \in \mathbb{F}_2^{2^\ell}$ and send $(\boldsymbol{H}, \boldsymbol{u} = \boldsymbol{H}\boldsymbol{e} + \boldsymbol{x})$ to $\mathscr{A}$.

Figure 5.3: Ideal functionality for leaky secret-shared two-party bit multiplication.

of zero is added to the outputs, note that the output distribution is uniformly random. Also, like IKNP, the protocol is *perfectly secure* against a corrupt P_A (or sender), so we only need to be concerned with leakage to a corrupt P_B who also sees the intermediate values of the protocol.

The leakage is different for each k, depending on whether $y_k = 0$ or $y_k = 1$, so we consider the two cases separately. Within each case, there are two potential sources of leakage: firstly, the corrupt P_B's knowledge of $\boldsymbol{t}_k$ and ρ_k may cause leakage (where ρ_k is a random share of zero), since these values are used to define P_A's output. Secondly, the d_k values seen by P_B, which equal

$$d_k = \mathsf{H}_{AB}(k, y_k, \boldsymbol{t}_k) \oplus \mathsf{H}_{AB}(k, 1 \oplus y_k, \boldsymbol{t}_k \oplus \Delta) \oplus x_k, \tag{5.1}$$

may leak information on P_A's inputs x_k.

Case 1 ($y_k = 1$)**:**

In this case there is only leakage from the values $\boldsymbol{t}_k$ and ρ_k, which are used to define P_A's output. Since $z_k^A = \mathsf{H}_{AB}(k, 0, \boldsymbol{t}_k \oplus \Delta) \oplus \rho_k$, all of P_A's outputs (and hence, also inputs) where $y_k = 1$ effectively have only ℓ bits of min-entropy in the view of P_B, corresponding to the random choice of Δ. In this case P_B's output is $z_k^B = z_k^A \oplus x_k = \mathsf{H}_{AB}(k, 0, \boldsymbol{t}_k \oplus \Delta) \oplus \rho_k \oplus x_k$. To ensure that P_B's view is simulable the functionality needs to sample a random string $\Delta \leftarrow \{0,1\}^\ell$ and leak $\mathsf{H}_{AB}(k, 0, \boldsymbol{t}_k \oplus \Delta) \oplus x_k$ to a corrupt P_B.

Concerning the d_k values, notice that when $y_k = 1$ P_B can compute $\mathsf{H}_{AB}(k, 1, \boldsymbol{t}_k)$ and use (5.1) to recover $\mathsf{H}_{AB}(k, 0, \boldsymbol{q}_k) + x_k$, which equals $z_k^A + \rho_k + x_k$. However, this is not a problem, because in this case we have $z_k^B = z_k^A + x_k$, so d_k can be simulated given P_B's output.

Case 2 ($y_k = 0$)**:**

Here the d_k values seen by P_B causes leakage on P_A's inputs, because Δ is short. Looking at (5.1), d_k leaks information on x_k because $\Delta \leftarrow \{0,1\}^\ell$ is the only unknown in the equation, and is fixed

Protocol $\Pi^{r,\ell}_{\text{Leaky-2-Mult}}$

PARAMETERS: r, number of multiplications; ℓ, key length.
INPUT: $\boldsymbol{x} = (x_1,\ldots,x_r) \in \mathbb{F}_2^r$ from P_A and $\boldsymbol{y} = (y_1,\ldots,y_r) \in \mathbb{F}_2^r$ from P_B.
COMMON INPUT: A hash function $\mathsf{H}_{AB} : [r] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}$.

1. P_A and P_B invoke $\mathscr{F}^{r,\ell}_{\Delta\text{-ROT}}$ where P_A is sender with a random input $\Delta \leftarrow \{0,1\}^\ell$, and P_B is receiver with inputs $(y_1,\ldots,y_r)$. P_A receives random strings $\boldsymbol{q}_k \in \{0,1\}^\ell$ and P_B receives $\boldsymbol{t}_k = \boldsymbol{q}_k \oplus y_k \cdot \Delta$, for $k \in [r]$.
2. Call $\mathscr{F}^r_{\text{Zero}}$ so that P_A and P_B obtain the same random $\rho_k \in \{0,1\}$ for every $k \in [r]$.
3. For each $k \in [r]$, P_A privately sends to P_B:
$$d_k = \mathsf{H}_{AB}(k,0,\boldsymbol{q}_k) + \mathsf{H}_{AB}(k,1,\boldsymbol{q}_k + \Delta) + x_k.$$
4. P_B outputs
$$z^B_k = \mathsf{H}_{AB}(k,y_k,\boldsymbol{t}_k) + y_k \cdot d_k + \rho_k, \quad \text{for } k \in [r].$$
5. P_A outputs
$$z^A_k = \mathsf{H}_{AB}(k,0,\boldsymbol{q}_k) + \rho_k, \quad \text{for } k \in [r].$$

Figure 5.4: Leaky secret-shared two-party bit multiplication protocol.

for every k. Similarly to the previous case, this means that all of P_A's inputs where $y_k = 0$ have only ℓ bits of min-entropy in the view of an adversary who corrupts P_B. We can again handle this leakage, by defining $\mathscr{F}_{\text{Leaky-2-Mult}}$ to leak $\mathsf{H}_{AB}(k,1,\boldsymbol{t}_k \oplus \Delta) + x_k$ to a corrupt P_B.

Note that there is no leakage from the $\boldsymbol{t}_k$ values when $y_k = 0$, because then $\boldsymbol{t}_k = \boldsymbol{q}_k$, so these messages are independent of Δ and the inputs of P_A.

In the functionality $\mathscr{F}_{\text{Leaky-2-Mult}}$, we actually modify the above slightly so that the leakage is defined in terms of linear algebra, instead of the hash function H_{AB}, to simplify the translation to the DRSD problem later on. Therefore, $\mathscr{F}_{\text{Leaky-2-Mult}}$ defines a matrix $\boldsymbol{H} \in \mathbb{F}_2^{r \times 2^\ell}$, which contains the 2^ℓ values $\{\mathsf{H}_{AB}(k,1 \oplus y_k, \boldsymbol{t}_k \oplus \Delta)\}_{\Delta \in \{0,1\}^\ell}$ in row k, where each $\boldsymbol{t}_k$ is uniformly random. Given $\boldsymbol{H}$, the leakage from the protocol can then be described by sampling a random unit vector $\boldsymbol{e} \in \mathbb{F}_2^{2^\ell}$ (which corresponds to $\Delta \in \{0,1\}^\ell$ in the protocol) and leaking $\boldsymbol{u} = \boldsymbol{H}\boldsymbol{e} + \boldsymbol{x}$ to a corrupt P_B.

5.3.1.4 Communication complexity and security

The cost of computing r secret-shared products is that of ℓ random, correlated OTs on r-bit strings, and a further r bits of communication. Using OT extension [10, 74] to implement the correlated OTs the amortized cost is $\ell(r+\kappa)$ bits, for computational security κ. This gives a total cost of $\ell(r+\kappa)+r$ bits.

Theorem 5.3.1. *Protocol* $\Pi^{r,\ell}_{\text{Leaky-2-Mult}}$ *securely implements the functionality* $\mathscr{F}^{r,\ell}_{\text{Leaky-2-Mult}}$ *with perfect security in the* $(\mathscr{F}_{\Delta\text{-ROT}}, \mathscr{F}_{\text{Zero}})$*-hybrid model in the presence of static honest-but-curious adversaries.*

Proof. The main challenge in the proof consists of showing that the leakage to P_B in the functionality can be translated directly to the leakage introduced in the protocol in the view of P_B. More formally, for the two cases of a corrupt P_A, and a corrupt P_B, we define a simulator who obtains the corrupted party's inputs and the output of $\mathcal{F}_{\text{Leaky-2-Mult}}$, and simulates the view of the corrupted party during a protocol execution.

NO CORRUPTIONS: Here, no simulation is necessary because all communication is over private channels, so we just need to show that the outputs of an honest execution are distributed identically to the functionality. By inspection, the protocol is correct. Observe that the outputs of P_A are uniformly random, because ρ_k is uniformly random. Since P_B's outputs are fixed by the inputs and P_A's outputs, we are done.

CORRUPT P_A: This is the simpler of the remaining two cases. The simulator $\mathcal{S}_A$ receives P_A's inputs $x_1,\ldots,x_r \in \mathbb{F}_2$, as well as the outputs $z_1^A,\ldots,z_r^A$ from $\mathcal{F}_{\text{Leaky-2-Mult}}$. It completes the view of P_A by sampling the $q_1,\ldots,q_r \leftarrow \{0,1\}^\ell$ P_A receives from $\mathcal{F}_{\Delta\text{-ROT}}$, and then sends $\rho_k = z_k^A - \mathsf{H}_{AB}(k,0,q_k)$ to simulate P_A's outputs from $\mathcal{F}_{\text{Zero}}$.

It is easy to see that the views in the two executions are identically distributed, since no messages are sent to P_A during the protocol, and the definition of ρ_k in the simulation ensures that ρ_k is uniformly random (because z_k^A is) and also consistent with P_A's output and the hash function, as in the protocol.

CORRUPT P_B: We define a simulator $\mathcal{S}_B$, who receives the inputs $y_1,\ldots,y_r \in \{0,1\}$, and then obtains the values $z_1^B,\ldots,z_r^B,\boldsymbol{H},\boldsymbol{u} = (u_1,\ldots,u_r)$ from the functionality.

Let $\mathcal{S}_B$ sample values $t_1,\ldots,t_r \in \{0,1\}^\ell$ at random, subject to the constraint that for every $k \in [r]$ and $k' \in \{0,1\}^\ell$, $\mathsf{H}_{AB}(k,1\oplus y_k, t_k \oplus k')$ is equal to entry (k,k') of $\boldsymbol{H}$ (viewing k' also as an integer in $[2^\ell]$). Note that because of the way $\boldsymbol{H}$ is defined in $\mathcal{F}_{\text{Leaky-2-Mult}}$, such a t_k is guaranteed to exist and can be found by searching all $2^{2\ell} = \text{poly}(\kappa)$ possibilities of k' and t_k. This also ensures it will be identically distributed to the t_k sampled by the functionality. $\mathcal{S}_B$ sends these values t_k as the outputs of $\mathcal{F}_{\Delta\text{-ROT}}$ to P_B.

For all $k \in [r]$, $\mathcal{S}_B$ then emulates the output of $\mathcal{F}_{\text{Zero}}$ to P_B as follows:

1. If $y_k = 0$, send $\rho_k = z_k^B + \mathsf{H}_{AB}(k,0,t_k)$.
2. If $y_k = 1$, send $\rho_k = z_k^B + u_k$.

Finally, for $k \in [r]$, $\mathcal{S}_B$ sends $d_k = u_k + \mathsf{H}_{AB}(k,y_k,t_k)$ to P_B. This completes the simulation of P_B's view.

Regarding indistinguishability, first note that, as observed above, the t_k values are identically distributed in both executions. Now considering the case when $y_k = 0$, we have:

$$z_k^B + \mathsf{H}_{AB}(k,0,t_k) + \rho_k = 0,$$

from the definition of ρ_k. Since in both worlds z_k^B, t_k and ρ_k are all uniformly random, subject to the above, this means that these values are identically distributed in both worlds. Also, it is easy

to see that the simulated d_k values are computed exactly as in the protocol, because of the way $\mathscr{F}_{\text{Leaky-2-Mult}}$ computes u_k.

When $y_k = 1$, we have:

$$\begin{aligned} z_k^B + \mathsf{H}_{AB}(k,1,t_k) + d_k + \rho_k = 0 &\iff (\rho_k + u_k) + \mathsf{H}_{AB}(k,1,t_k) + \rho_k = d_k \\ &\iff \mathsf{H}_{AB}(k,0,t_k \oplus \tilde{\Delta}) + \mathsf{H}_{AB}(k,1,t_k) + x_k = d_k, \end{aligned}$$

where $\tilde{\Delta} \in [2^\ell]$ denotes the position of the 1 in $\boldsymbol{e}$ sampled by $\mathscr{F}_{\text{Leaky-2-Mult}}$ to compute $\boldsymbol{u}$, so is identically distributed to $\Delta \in \{0,1\}^\ell$ in the real protocol. Therefore, the last equation above holds, which implies that z_k^B, ρ_k and d_k are all distributed identically to the values in the real protocol. ■

5.3.2 MPC for Binary Circuits From Leaky OT

We now show how to use the leaky OT protocol to compute multiplication triples over $\mathbb{F}_2$, using a GMW-style protocol [62, 64] optimized for the case of at least h honest parties. This can then be used to obtain a general MPC protocol for binary circuits using Beaver's method [16].

5.3.2.1 Triple generation

We implement the triple generation functionality over $\mathbb{F}_2$, shown in Figure 5.5. Recall that to create a triple using the GMW method, first each party locally samples shares $x^i, y^i \leftarrow \mathbb{F}_2$. Next, the parties compute shares of the product based on the fact that

$$(\sum_{i=1}^n x^i) \cdot (\sum_{i=1}^n y^i) = \sum_{i=1}^n x^i y^i + \sum_{i=1}^n \sum_{j \neq i} x^i y^j,$$

where x^i denotes P_i's share of $x = \sum_i x^i$.

Since each party can compute $x^i y^i$ on its own, in order to obtain additive shares of $z = xy$ it suffices for the parties to obtain additive shares of $x^i y^j$ for every pair $i \neq j$. This is done using oblivious transfer between P_i and P_j, since a 1-out-of-2 OT implies two-party secret-shared bit multiplication. Due to efficiency considerations we realize a slight variation of this functionality where two (possibly overlapping) subsets $\mathscr{P}_{(h)}, \mathscr{P}_{(1)}$ such that $\mathscr{P}_{(h)}$ has at least h honest parties and $\mathscr{P}_{(1)}$ has at least one honest party, choose the respective shares of x and y.

If we use the *leaky* two-party batch multiplication protocol from the previous section, this approach fails to give a secure protocol because the leakage in $\mathscr{F}_{\text{Leaky-2-Mult}}$ allows a corrupt P_B to guess P_A's inputs with probability $2^{-\ell}$. When using this naively, P_A carries out a secret-shared multiplication using the *same input shares* with every other party, which allows every corrupt party to attempt to guess P_A's shares, increasing the success probability further. If the number of corrupted parties is not too small then this gives the adversary a significant chance of successfully guessing the shares of *every honest party*, completely breaking security.

Functionality $\mathscr{F}^r_{\text{Triple}}$

1. Sample $(x^i_j, y^i_j, z^i_j) \leftarrow \mathbb{F}_2^3$, for $i \in [n]$ and $j \in [r]$, subject to the constraint that

$$\sum_i z^i_j = \left(\sum_i x^i_j\right) \cdot \left(\sum_i y^i_j\right)$$

2. Output (x^i_j, y^i_j, z^i_j) to party P_i, for $j \in [r]$.

Figure 5.5: Multiplication triple generation functionality.

To avoid this issue, we require P_A to *randomize* the shares used as input to $\mathscr{F}_{\text{Leaky-2-Mult}}$, in such a way that we still preserve correctness of the protocol. To do this, the parties will use $\mathscr{F}_{\text{Zero}}$ to generate random zero shares $s^{i,j} \in \mathbb{F}_2$ (held by P_i), satisfying $\sum_i s^{i,j} = 0$ for all $j \in [n]$, and then P_i and P_j will multiply $x^i + s^{i,j}$ and y^j. This means that all parties end up computing shares of

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(x^i + s^{i,j})y^j = \sum_{j=1}^{n} y^j \sum_{i=1}^{n}(x^i + s^{i,j}) = xy,$$

so still obtain a correct triple.

Finally, to ensure that the output shares are uniformly random, fresh shares of zero will be added to each party's share of xy. Note that masking each x^i input to $\mathscr{F}_{\text{Leaky-2-Mult}}$ means that it doesn't matter if the individual shares are leaked to the adversary, as long as it is still hard to guess the *sum of all shares*. This means that we only need to be concerned with the *sum of the leakage* from $\mathscr{F}_{\text{Leaky-2-Mult}}$. Recall that each individual instance leaks the input of an honest party P_i masked by $\boldsymbol{H}_i\boldsymbol{e}_i$, where $\boldsymbol{H}_i$ is a random matrix and $\boldsymbol{e}_i \in \mathbb{F}_2^{2^\ell}$ is a random unit vector. Summing up all the leakage from h honest parties, we get

$$\sum_{i=1}^{h} \boldsymbol{H}_i\boldsymbol{e}_i = (\boldsymbol{H}_1 \| \cdots \| \boldsymbol{H}_h)\begin{pmatrix}\boldsymbol{e}_1\\ \vdots \\ \boldsymbol{e}_h\end{pmatrix}$$

This is exactly an instance of the $\mathsf{DRSD}_{r,h,\ell}$ problem, so is pseudorandom for an appropriate choice of parameters.

We remark that the number of triples generated, r, affects the hardness of DRSD. However, we can create an arbitrary number of triples without changing the assumption by repeating the protocol for a fixed r.

5.3.2.2 Reducing the number of OT channels

The above approach reduces communication of GMW by a factor κ/ℓ, for ℓ-bit keys, but still requires a complete network of $n(n-1)$ OT and communication channels between the parties. We can reduce this further by again taking advantage of the fact that there are at least h honest parties. We observe that when using our two-party secret-shared multiplication protocol

Protocol Π^r_{Triple}

The protocol runs between a set of parties $\mathscr{P} = \{P_1, \ldots, P_n\}$, containing two (possibly overlapping) subsets $\mathscr{P}_{(h)}, \mathscr{P}_{(1)}$, such that $\mathscr{P}_{(h)}$ has at least h honest parties and $\mathscr{P}_{(1)}$ has at least one honest party. We denote $n_h = |\mathscr{P}_{(h)}|$, $n_1 = |\mathscr{P}_{(1)}|$.

CRS: Random hash functions $\mathsf{H}_i : [r] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}$, for $i \in [n_h]$.

1. Each party $P_i \in \mathscr{P}_{(h)}$ samples $x^i_k \leftarrow \mathbb{F}_2$, and each $P_j \in \mathscr{P}_{(1)}$ samples $y^j_k \leftarrow \mathbb{F}_2$, for $k \in [r]$.
2. Call $\mathscr{F}_{\text{Zero}}$ so that each $P_\tau \in \mathscr{P}_{(h)} \cup \mathscr{P}_{(1)}$ obtains shares $(\rho^\tau_1, \ldots, \rho^\tau_r)$ and each $P_i \in \mathscr{P}_{(h)}$ obtains shares $(s^{i,j}_1, \ldots, s^{i,j}_r)_{j \in \mathscr{P}_{(1)}}$, such that $\bigoplus_{\tau \in \mathscr{P}_{(h)} \cup \mathscr{P}_{(1)}} \rho^\tau_k = 0$ and $\bigoplus_{i \in \mathscr{P}_{(h)}} s^{i,j}_k = 0$.
3. Every pair $(P_i, P_j) \in \mathscr{P}_{(h)} \times \mathscr{P}_{(1)}$ runs $\mathscr{F}^{r,\ell}_{\text{Leaky-2-Mult}}(\mathsf{H}_i)$ on input $\{x^i_k + s^{i,j}_k\}_{k \in [r]}$ from P_i and $\{y^j_k\}_{k \in [r]}$ from P_j. For $k \in [r]$, P_i receives $a^{i,j}_k$ and P_j receives $b^{j,i}_k$ such that $a^{i,j}_k + b^{j,i}_k = (x^i_k + s^{i,j}_k) \cdot y^j_k$.
4. Each $P_i \in \mathscr{P}_{(h)} \cup \mathscr{P}_{(1)}$ computes, for $k \in [r]$:

$$z^i_k = (x^i_k + s^{i,i}_k) \cdot y^i_k + \sum_{j \neq i} (a^{i,j}_k + b^{i,j}_k) + \rho^i_k$$

   where if any value $x^i_k, y^i_k, a^{i,j}_k, b^{i,j}_k, s^{i,i}_k$ has not been defined by P_i, it is set to zero.
5. P_i outputs the shares $(x^i_k, y^i_k, z^i_k)_{k \in [r]}$.

Figure 5.6: Secret-shared triple generation using leaky two-party multiplication.

to generate triples, information is only leaked on the x^i shares, and not the y^i shares of each triple. This means that $h-1$ parties can choose their shares of y to be zero, and y will still be uniformly random to an adversary who corrupts up to $t = n - h$ parties. This reduces the number of OT channels needed from $n(n-1)$ to $(t+1)(n-1)$.

When the number of parties is large enough, we can do even better using *random committees*. We randomly choose two committees, $\mathscr{P}_{(h)}$ and $\mathscr{P}_{(1)}$, such that except with negligible probability, $\mathscr{P}_{(h)}$ has at least h honest parties and $\mathscr{P}_{(1)}$ has at least one honest party. Only the parties in $\mathscr{P}_{(h)}$ choose non-zero shares of x, and parties in $\mathscr{P}_{(1)}$ choose non-zero shares of y; all other parties do not take part in any OT instances, and just output random sharings of zero. We remark that it can be useful to choose the parameter h *lower than* the actual number of honest parties, to enable a smaller committee size (at the cost of potentially larger keys). When the total number of parties, n, is large enough, this means the number of interacting parties can be independent of n. The complete protocol, described for two fixed committees satisfying our requirements, is shown in Figure 5.6.

5.3.2.3 Communication complexity and security

Recall from the analysis in Section 5.3.1 that when using protocol $\Pi_{\text{Leaky-2-Mult}}$ with Π_{Triple}, the cost of computing r secret-shared triples is that of ℓ random, correlated OTs on r-bit strings, and a further r bits of communication between every pair of parties. This gives a total cost of

$\ell(r+\kappa)+r$ bits between every pair of parties who has an OT channel (ignoring $\mathcal{F}_{\text{Zero}}$ and the seed OTs for OT extension, since their communication cost is independent of the number of triples). If the two committees $\mathcal{P}_{(h)}, \mathcal{P}_{(1)}$ have sizes $n_h \leq n$ and $n_1 \leq t+1$ then we have the following theorem.

Theorem 5.3.2. *Protocol Π_{Triple} securely realizes $\mathcal{F}_{\text{Triple}}^{r}$ in the $(\mathcal{F}_{\text{Leaky-2-Mult}}^{r,\ell}, \mathcal{F}_{\text{Zero}}^{(n+1)r})$-hybrid model, based on the $\mathsf{DRSD}_{r,h,\ell}$ assumption, where h is the number of honest parties in $\mathcal{P}_{(h)}$. The amortized communication cost is $\leq n_h n_1(\ell + \ell\kappa/r + 1)$ bits per triple.*

Proof. The claimed communication complexity follows from the previous analysis. Security relies on the fact that $P_i \in \mathcal{P}_{(h)}$'s input to $\mathcal{F}_{\text{Leaky-2-Mult}}$ is always of the form $x^i + s^{i,j}$, where $s^{i,j}$ is a fresh, random sharing of zero. This means that any leakage on P_i's input from $\mathcal{F}_{\text{Leaky-2-Mult}}$ is perfectly masked by $s^{i,j}$, and we only need to consider the *sum* of the leakage from all honest parties in $\mathcal{P}_{(h)}$.

Recall that we have two committees $\mathcal{P}_{(h)}$ and $\mathcal{P}_{(1)}$ of sizes n_h and n_1, with at least h and 1 honest parties, respectively. Let $\mathcal{A}$ be an adversary corrupting a set of parties A. Throughout the proof we will write $x_1, \ldots, x_r$ to denote the components of a vector $\boldsymbol{x} \in \mathbb{F}_2^r$.

We construct a simulator, $\mathcal{S}$, which interacts with $\mathcal{A}$ as follows:

1. Simulate the CRS with n_h randomly sampled functions $\mathsf{H}_i : [r] \times \{0,1\} \times \{0,1\}^\ell \to \{0,1\}$.
2. Call $\mathcal{F}_{\text{Triple}}$ to receive the corrupted parties' outputs, $(x_k^i, y_k^i, z_k^i)_{i \in A \cap (\mathcal{P}_{(h)} \cup \mathcal{P}_{(1)}), k \in [r]}$.
3. For each $i \in \mathcal{P}_{(h)} \cap A$, sample $\boldsymbol{s}^{i,j} \leftarrow \mathbb{F}_2^r$, for $j \in [n_1]$, and send these to $\mathcal{A}$ as the shares output by $\mathcal{F}_{\text{Zero}}$.
4. Let $P_i \in \mathcal{P}_{(h)}, P_j \in \mathcal{P}_{(1)}$. Compute the messages that would be sent by $\mathcal{F}_{\text{Leaky-2-Mult}}$ to the adversary as follows:
   a) $P_i, P_j \in A$: Using both parties' inputs, generate their random output shares as $\mathcal{F}_{\text{Leaky-2-Mult}}$ would do and send these to $\mathcal{A}$. Explicitly, the simulator samples shares $\boldsymbol{a}^{i,j}, \boldsymbol{b}^{j,i}$ that sum to $(\boldsymbol{x}^i + \boldsymbol{s}^{i,j}) * \boldsymbol{y}^j$, and the leakage $(\boldsymbol{H}^{i,j}, \boldsymbol{u}^{i,j})$ on $\boldsymbol{x}^i + \boldsymbol{s}^{i,j}$ (just as $\mathcal{F}_{\text{Leaky-2-Mult}}$ would do).
   b) $P_i \in A, P_j \notin A$: Emulate the corrupt P_i's view honestly, by sampling $\boldsymbol{a}^{i,j} \leftarrow \mathbb{F}_2^r$.
   c) $P_i \notin A, P_j \in A$: Sample uniform values $\boldsymbol{b}^{j,i}, \boldsymbol{u}^{i,j} \leftarrow \mathbb{F}_2^r$, and sample $\boldsymbol{H}^{i,j} \in \mathbb{F}_2^{m \times 2^\ell}$ exactly as $\mathcal{F}_{\text{Leaky-2-Mult}}$ would do, using knowledge of H_i and $\boldsymbol{y}^j$.
5. For $i \in A \cap (\mathcal{P}_{(h)} \cup \mathcal{P}_{(1)})$, compute $\boldsymbol{\rho}^i = \boldsymbol{z}^i + (\boldsymbol{x}^i + \boldsymbol{s}^{i,i}) * \boldsymbol{y}^i + \sum_{j \neq i} (\boldsymbol{a}^{i,j} + \boldsymbol{b}^{i,j})$ and send this as the ρ_k^i share from $\mathcal{F}_{\text{Zero}}$.
6. Send to $\mathcal{A}$ the values $\{\boldsymbol{a}^{i,j}\}_{i \in \mathcal{P}_{(h)} \cap A, j \in \mathcal{P}_{(1)}} \cup \{\boldsymbol{b}^{j,i}, \boldsymbol{H}^{i,j}, \boldsymbol{u}^{i,j}\}_{i \in \mathcal{P}_{(h)}, j \in \mathcal{P}_{(1)} \cap A}$ as defined above, to simulate the outputs of $\mathcal{F}_{\text{Leaky-2-Mult}}$.

We first consider the distribution of the parties' outputs.

Claim 5.3.1. *The outputs of the protocol are distributed identically to outputs of the functionality.*

Proof. We need to to show that, in the real protocol, $\{z_k^i\}_{i,k}$ are uniformly random subject to $\sum_i z_k^i = \sum_i x_k^i \cdot \sum_i y_k^i$. Firstly, the correctness constraint holds because

$$\begin{aligned}
\sum_{i=1}^{n} z_k^i &= \sum_{i=1}^{n} \left((x_k^i + s_k^{i,i}) \cdot y_k^i + \sum_{j \neq i} (a_k^{i,j} + b_k^{i,j}) + \rho_k^i \right) \\
&= \sum_{i=1}^{n} x_k^i \cdot y_k^i + \sum_{i=1}^{n} \left(y_k^i \cdot s_k^{i,i} + \sum_{j \neq i} (a_k^{i,j} + b_k^{j,i}) \right) \\
&= \sum_{i=1}^{n} x_k^i \cdot y_k^i + \sum_{i=1}^{n} \left(y_k^i \cdot s_k^{i,i} + \sum_{j \neq i} y_k^i \cdot (x_k^j + s_k^{j,i}) \right) \\
&= x_k \cdot y_k + \sum_{i=1}^{n} y_k^i \cdot \sum_{j=1}^{n} s_k^{j,i} \\
&= x_k \cdot y_k
\end{aligned}$$

where the second line above holds because $\sum_i \rho_k^i = 0$, and the final line uses $\sum_j s_k^{j,i} = 0$.

Now, to see that $(z_k^i)_i$ are *uniformly random*, subject to the above, notice that the masks $(\rho_k^i)_{i=1}^{n-1}$ are uniformly random in the protocol, so the same is true of $(z_k^i)_{i=1}^{n-1}$. This completes the claim. □

We next consider the entire view of the environment $\mathcal{Z}$, which is the joint distribution of all parties' inputs and outputs, and the messages received by the adversary during the protocol. Using vector notation, this is:

$$(\boldsymbol{x}^i, \boldsymbol{y}^j, \boldsymbol{z}^i, \boldsymbol{z}^j)_{i \in \mathscr{P}_{(h)}, j \in \mathscr{P}_{(1)}}, (\boldsymbol{\rho}^i, \boldsymbol{s}^{i,j}, \boldsymbol{a}^{i,j})_{i \in A \cap \mathscr{P}_{(h)}, j \in \mathscr{P}_{(1)}}, (\boldsymbol{\rho}^j, \boldsymbol{b}^{j,i}, \boldsymbol{u}^{i,j}, \boldsymbol{H}^{i,j})_{i \in \mathscr{P}_{(h)}, j \in \mathscr{P}_{(1)} \cap A}.$$

First note that the $\boldsymbol{\rho}^\tau, \tau \in \mathscr{P}_{(h)} \cup \mathscr{P}_{(1)}$ and $\boldsymbol{s}^{i,j}$ shares, for $i \in \mathscr{P}_{(h)} \cap A, j \in \mathscr{P}_{(1)} \setminus A$, are uniformly random in both executions, since the environment never sees the honest parties' shares. Secondly, recall that in the simulation, $\boldsymbol{a}^{i,j}$ for corrupt P_i and honest P_j and $\boldsymbol{b}^{j,i}$ (for corrupt P_j and honest P_i) are computed uniformly at random, and this is identically distributed to the values in the protocol sampled by $\mathscr{F}_{\text{Leaky-2-Mult}}$, because the outputs of the honest party in that instance are not seen by $\mathscr{Z}$. Also, notice that when P_j is corrupt $\mathscr{S}$ computes $\boldsymbol{H}^{i,j}$ exactly as in the real protocol, because $\mathscr{S}$ knows P_j's input $\boldsymbol{y}^j$.

This leaves the $\{\boldsymbol{u}^{i,j}\}_{i \in \mathscr{P}_{(h)} \setminus A, j \in \mathscr{P}_{(1)} \cap A}$ values, which are the main challenge, because the simulation computes these with random values, whilst the real execution uses the honest P_i's inputs, computing $\boldsymbol{u}^{i,j} = \boldsymbol{H}^{i,j} \boldsymbol{e}^{i,j} + \boldsymbol{x}^i + \boldsymbol{s}^{i,j}$ for a random unit vector $\boldsymbol{e}^{i,j}$. Let $P_{i_1}, \ldots, P_{i_h}$ be the honest parties in $\mathscr{P}_{(h)}$. Because the $\boldsymbol{s}^{i,j}$ values are random shares of zero, it holds that the partial views containing the entire transcript *except for* $(\boldsymbol{u}^{i_1,j})_{j \in \mathscr{P}_{(1)} \cap A}$ are identically distributed. This is because for $P_j \in \mathscr{P}_{(1)} \cap A$, the masks $\boldsymbol{s}^{i_2,j}, \ldots, \boldsymbol{s}^{i_h,j}$ in the protocol are random and independent of the view of $\mathscr{Z}$, which makes the corresponding $\boldsymbol{u}^{i_2,j}, \ldots, \boldsymbol{s}^{i_h,j}$ values distributed the same as in the simulation.

Once we include $\boldsymbol{u}^{i_1,j}$, however, these values are no longer independent because $\sum_{i\in\mathscr{P}_{(h)}}^{n}\boldsymbol{s}^{i,j}=0$. We therefore look at the distribution of $\sum_{k=1}^{h}\boldsymbol{u}^{i_k,j}$, for some fixed $j\in\mathscr{P}_{(1)}\cap A$. In the protocol, we have

$$\sum_{i\in\mathscr{P}_{(h)}\setminus A}\boldsymbol{u}^{i,j}=\sum_{i\in\mathscr{P}_{(h)}\setminus A}(\boldsymbol{x}^i+\boldsymbol{s}^{i,j}+\boldsymbol{H}^{i,j}\boldsymbol{e}^{i,j})$$

for some random, weight-1 vector $\boldsymbol{e}^{i,j}$. In the simulation, all of the $\boldsymbol{u}^{i,j}$'s are uniformly random.

Since $\mathscr{Z}$ can compute $\sum_{i\in\mathscr{P}_{(h)}\setminus A}(\boldsymbol{x}^i+\boldsymbol{s}^{i,j})$ with the information it already has, it follows that distinguishing the two executions requires distinguishing

$$\boldsymbol{H}_i\boldsymbol{e}_i:=\left(\boldsymbol{H}^{i_1,j}\|\boldsymbol{H}^{i_2,j}\|\cdots\|\boldsymbol{H}^{i_h,j}\right)\begin{pmatrix}\boldsymbol{e}^{i_1,j}\\ \boldsymbol{e}^{i_2,j}\\ \vdots\\ \boldsymbol{e}^{i_h,j}\end{pmatrix}$$

and the uniform distribution on r bits (given $\boldsymbol{H}_i$).

We claim that this corresponds exactly to solving the $\mathsf{DRSD}_{r,h,\ell}$ problem, because $\boldsymbol{H}_i$ is uniformly distributed in $\mathbb{F}_2^{r\times 2^\ell h}$ and $\boldsymbol{e}_i$ is a uniformly random, 1-regular error vector of weight h.

Lemma 5.3.2. *Any environment distinguishing the real and ideal executions with advantage δ can be used to break $\mathsf{DRSD}_{r,h,\ell}$ with advantage at least δ/t (where $t=|\mathscr{P}_{(1)}\cap A|$).*

Proof. Assume w.l.o.g. that the corrupted parties in $\mathscr{P}_{(1)}$ are indexed $P_1,\ldots,P_t$. We construct a sequence of hybrid executions, $\mathbf{HYB}_0,\ldots,\mathbf{HYB}_t$, where hybrid $\mathbf{HYB}_0$ is identical to the simulation. In hybrid $\mathbf{HYB}_{j'}$, instead of the simulator sampling $\boldsymbol{u}^{i,j}$ (for $j\le j', i\in\mathscr{P}_{(h)}\setminus A$) at random, we replace this with the real $\boldsymbol{u}^{i,j}$ generated using P_i's inputs as in the protocol. The final hybrid $\mathbf{HYB}_t$ is therefore identically distributed to the real execution.

Let $\mathscr{A}$ be an adversary for which the environment $\mathscr{Z}$ distinguishes between $\mathbf{HYB}_{j'}$ and $\mathbf{HYB}_{j'+1}$ with advantage δ, for some index $j'<t$. We construct a distinguisher $\mathscr{D}$ for $\mathsf{DRSD}_{r,h,\ell}$ as follows. $\mathscr{D}$ receives a DRSD challenge $\boldsymbol{H}_{j'}\in\mathbb{F}_2^{r\times h2^\ell}, \boldsymbol{c}^{j'}\in\mathbb{F}_2^r$. Write $\boldsymbol{H}_{j'}=(\boldsymbol{H}^{i_1,j'}\|\boldsymbol{H}^{i_2,j'}\|\cdots\|\boldsymbol{H}^{i_h,j'})$, where each $\boldsymbol{H}^{i_k,j'}\in\mathbb{F}_2^{r\times 2^\ell}$. Now $\mathscr{D}$ simulates an execution of Π_{Triple} with $\mathscr{A}$ as $\mathscr{S}$ would, with the following differences.

- $\mathscr{D}$ samples a set of honest parties' shares, $(\boldsymbol{x}^i,\boldsymbol{y}^i,\boldsymbol{z}^i)_{i\notin A}$ which, together with the corrupt parties' shares known to $\mathscr{D}$, form correct triples.
- Instead of sampling the function H_i in the CRS at random, sample it such that for every $k\in[h]$, the matrix $\boldsymbol{H}^{i_k,j'}$, sent later to the corrupt $P_{j'}$, is equal to the challenge matrix $\boldsymbol{H}^{i_k,j'}$. (The remainder of the CRS is sampled at random.)
- Instead of sampling the leakage terms $\boldsymbol{u}^{i_k,j'}$ (for $k\in[h]$) uniformly and independently, sample them at random such that $\sum_{i\in\mathscr{P}_{(h)}}(\boldsymbol{u}^{i,j'}+\boldsymbol{x}^i)=\boldsymbol{c}^{j'}$.

- For each $j < j'$, instead of sampling $\boldsymbol{u}^{i_k,j'}$ uniformly, compute them as in the real protocol using the honest parties' shares and shares of zero.

To conclude, $\mathscr{D}$ sends all the output shares to $\mathscr{Z}$ and outputs the same as $\mathscr{Z}$.

If the challenge $(\boldsymbol{H}_{j'}, \boldsymbol{c}^{j'})$ comes from the DRSD distribution then the $\boldsymbol{u}^{i_k,j'}$ values are distributed as in a real execution, so we are in hybrid $\mathbf{HYB}_{j'+1}$. On the other hand, if $\boldsymbol{c}^{j'}$ is uniformly random then so are the $\boldsymbol{u}^{i_k,j'}$, so we are in $\mathbf{HYB}_{j'}$. Therefore, the advantage of $\mathscr{D}$ is δ, the same as that of $\mathscr{Z}$. A standard hybrid argument then shows that there exists a distinguisher for $\mathbf{HYB}_0$ and $\mathbf{HYB}_t$, which has advantage at least δ/t. □

This concludes the proof of Theorem 5.3.2. ■

5.3.2.4 Parameters for unconditional security

Recall from Lemma 5.2.1 and Corollary 5.2.1 that if $\ell = 1$ and $h \geq r + s$, or if $h \geq (r + 2s)/\ell$ for any ℓ, then $\mathsf{DRSD}_{r,h,\ell}$ is *statistically hard*, with statistical security 2^{-s}. This means when h is large enough we can use 1-bit keys, and every pair of parties who communicates only needs to send $2 + \kappa/r$ bits over the network.[2]

5.3.2.5 MPC using multiplication triples

Our protocol for multiplication triples can be used to construct a semi-honest MPC protocol for binary circuits using Beaver's approach [16]. The parties first secret-share their inputs between all other parties. Then, XOR gates can be evaluated locally on the shares, whilst an AND gate requires consuming a multiplication triple, and two openings with Beaver's method. Each opening can be done with $2(n-1)$ bits of communication as follows: all parties send their shares to P_1, who sums the shares together and sends the result back to every other party.

In the 1-bit key case mentioned above, using two (deterministic) committees of sizes n and $t+1$ and setting, for instance, $r = \kappa$ implies the following corollary. Note that the number of communication channels is $(t+1)(n-1)$ and not $(t+1)n$, because in the deterministic case $\mathscr{P}_{(1)}$ is contained in $\mathscr{P}_{(h)}$, so $t+1$ sets of the shared cross-products can be computed locally.

Corollary 5.3.3. *Assuming OT and OWF, there is a semi-honest MPC protocol for binary circuits with an amortized communication complexity of no more than* $3(t+1)(n-1) + 4(n-1)$ *bits per AND gate, if there are at least* $\kappa + s$ *honest parties.*

5.4 Multi-Party Garbled Circuits with Short Keys

In this section we present our second contribution: a constant-round MPC protocol based on garbled circuits with short keys. The protocol has two phases, a preprocessing phase independent

[2] Note that we still need computational assumptions for OT and zero sharing in order to implement $\mathscr{F}_{\text{Leaky-2-Mult}}$ and $\mathscr{F}_{\text{Zero}}$.

of the parties' actual inputs where the garbled circuit is mutually generated by all parties, and an online phase where the computation is performed. We first abstractly discuss the details of our garbling method, and then turn to the two protocols for generating and evaluating the garbled circuit.

5.4.1 The Multi-Party Garbling Scheme

Our garbling method is defined by the functionality $\mathscr{F}_{\text{Preprocessing}}$ (Figure 5.7), which creates a garbled circuit that is given to all the parties. It can be seen as a variant of the multi-party garbling technique by Beaver, Micali and Rogaway [18], known as BMR and described in Section 2.6.

We recall that the main ideas behind most protocols instantiating the paradigm in Section 2.6. First, every party P_i contributes a pair of keys $k^i_{w,0}, k^i_{w,1} \in \{0,1\}^\kappa$ and a share of a wire mask $\lambda^i_w \in \{0,1\}$ for each wire w in the circuit. To garble a gate, the corresponding output wire key from every party is 'double encrypted' under the combination of all parties' input wire keys, using a PRF or PRG, so that no single party knows all the keys for a gate. In addition, the Free-XOR property can be supported by having each party choose their keys such that $k^i_{w,0} \oplus k^i_{w,1} = \Delta^i$, where Δ^i is a global fixed random string known to P_i.

The main difference between our work and the ones previously discussed in Section 2.6, Chapter 3 and Chapter 4 is that we use short keys of length ℓ_{BMR} instead of κ, and then garble gates using a random, expanding function $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^{\ell_{\text{BMR}}} \to \{0,1\}^{n\ell_{\text{BMR}}+1}$. Instead of basing security on a PRF or PRG, we then reduce the security of the protocol to the pseudorandomness of the *sum* of H when applied to each of the honest parties' keys, which is implied by the DRSD problem from Section 5.2.4. We also use H' to denote H with the least significant output bit dropped, which we use for garbling splitter gates.

To garble an AND gate g with input wires u, v and output wire w, each of the 4 garbled rows $\tilde{g}_{a,b}$, for $(a,b) \in \{0,1\}^2$, is computed as

$$\tilde{g}_{a,b} = \left(\bigoplus_{i=1}^{n} \mathsf{H}(i, b, k^i_{u,a}) \oplus \mathsf{H}(i, a, k^i_{v,b}) \right) \oplus (c, k^1_{w,c}, \ldots, k^n_{w,c}), \tag{5.2}$$

where $c = (a \oplus \lambda_u) \cdot (b \oplus \lambda_v) \oplus \lambda_w$ and $\lambda_u, \lambda_v, \lambda_w$ are the secret-shared wire masks. Each row can be seen as an encryption of the correct n output wire keys under the corresponding input wire keys of all parties. Note that, for each wire, P_i holds the keys $k^i_{u,0}, k^i_{u,1}$ and an additive share λ^i_u of the wire mask. The extra bit value that H takes as input is added to securely increase the stretch of H when using the same input key twice, preventing a 'mix-and-match' attack on the rows of a garbled gate. The output of H is also extended by an extra bit, to allow encryption of the output wire mask c.[3]

[3]This only becomes necessary when using short keys – in BMR with full-length keys the parties can recover the wire mask by comparing the output with their own two keys, but this does not work if collisions are possible.

Functionality $\mathcal{F}_{\text{Preprocessing}}$

COMMON INPUT: A function $\mathsf{H}:[n]\times\{0,1\}\times\{0,1\}^{\ell_{\text{BMR}}}\to\{0,1\}^{n\ell_{\text{BMR}}+1}$. Let H' denote the same function excluding the least significant bit of the output.

Let C_f be a boolean circuit with fan-out-one gates. Denote by $\mathsf{AND},\mathsf{XOR}$ and SPLIT its sets of AND, XOR and Splitter gates, respectively. Given a gate, let I and O be the set of its input and output wires, respectively. If $g\in\mathsf{SPLIT}$, then $I=\{w\}$ and $O=\{u,v\}$, otherwise $O=\{w\}$.
The functionality proceeds as follows $\forall i\in[n]$:

1. $\forall g\in\mathsf{XOR}$, sample $\Delta^i_g\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$.
2. For each circuit-input wire u, sample $\lambda_u\leftarrow\{0,1\}$ and $k^i_{u,0}\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$. If u is input to a XOR gate g, set $k^i_{u,1}=k^i_{u,0}\oplus\Delta^i_g$, otherwise $k^i_{u,1}\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$.
3. Passing topologically through all the gates $g\in\{\mathsf{AND}\cup\mathsf{XOR}\cup\mathsf{SPLIT}\}$ of the circuit:
   - If $g\in\mathsf{XOR}$:
     - Set $\lambda_w=\bigoplus_{x\in I}\lambda_x$
     - Set $k^i_{w,0}=\bigoplus_{x\in I}k^i_{x,0}$ and $k^i_{w,1}=k^i_{w,0}\oplus\Delta^i_g$
   - If $g\in\mathsf{AND}$:
     - Sample $\lambda_w\leftarrow\{0,1\}$.
     - $k^i_{w,0}\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$. If w is input to a XOR gate g' set $k^i_{w,1}=k^i_{w,0}\oplus\Delta^i_{g'}$, else $k^i_{w,1}\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$.
     - For $a,b\in\{0,1\}$, representing the external values of wires u and v, let $c=(a\oplus\lambda_u)\cdot(b\oplus\lambda_v)\oplus\lambda_w$. Store the four entries of the garbled version of g as:
$$\tilde{\boldsymbol{g}}_{a,b}=\left(\bigoplus_{i=1}^{n}\mathsf{H}(i,b,k^i_{u,a})\oplus\mathsf{H}(i,a,k^i_{v,b})\right)\oplus(c,k^1_{w,c},\ldots,k^n_{w,c}),\quad(a,b)\in\{0,1\}^2.$$
   - If $g\in\mathsf{SPLIT}$:
     - Set $\lambda_x=\lambda_w$ for every $x\in O$.
     - $\forall x\in O$, sample $k^i_{x,0}\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$. If $x\in O$ is input to a XOR gate g', set $k^i_{x,1}=k^i_{x,0}\oplus\Delta^i_{g'}$, otherwise $k^i_{x,1}\leftarrow\{0,1\}^{\ell_{\text{BMR}}}$.
     - For $c\in\{0,1\}$, the external value on w, store the two entries of the garbled version of g as:
$$\tilde{g}_c=\left(\bigoplus_{i=1}^{n}\mathsf{H}'(i,0,k^i_{w,c}),\bigoplus_{i=1}^{n}\mathsf{H}'(i,1,k^i_{w,c})\right)\oplus(k^1_{u,c},\ldots,k^n_{u,c},k^1_{v,c},\ldots,k^n_{v,c}),\ c\in\{0,1\}$$
4. **Output:** For each circuit-input wire u, send λ_u to the party providing inputs to C_f on u. For every circuit wire v and $i\in[n]$, send $k^i_{v,0},k^i_{v,1}$ to P_i. Finally, send to all parties $\tilde{g}$ for each $g\in\mathsf{AND}\cup\mathsf{SPLIT}$ and λ_w for each circuit-output wire w.

Figure 5.7: Multi-party garbling functionality.

5.4.1.1 Splitter gates

When relying on the DRSD problem, the reuse of a key in multiple gates degrades parameters and makes the problem easier (as the parameter r grows, the key length must be increased), so we cannot handle circuits with arbitrary fan-out. For this reason, we restrict our exposition of the garbling to fan-out-one circuits with so-called *splitter gates*. This type of gate takes as input a single wire w and provides two output wires u, v, each of them with fresh, independent keys representing the same value carried by the input wire. Converting an arbitrary circuit to use splitter gates incurs a cost of roughly a factor of two in the circuit size (see below).

Splitter gates were previously introduced in [126] as a fix for a similar issue in the original BMR paper [18], where the wire 'keys' were used as seeds for a PRG in order to garble the gates, so that when a wire was used as input to multiple gates, their garbled versions did not use independent pseudorandom masks. Other recent BMR-style papers avoid this issue by applying the PRF over the gate identifier as well, which produces distinct, independent PRF evaluations for each gate.

5.4.1.2 Enabling Free-XOR

The Free-XOR [86] optimization results in an improvement in both computation and communication for XOR gates where a global fixed random Δ_i is chosen by each party P_i and the input keys are locally XORed, yielding the output key of this gate. We cannot use the standard Free-XOR technique [25, 86] for the same reason discussed above: reusing a single offset across multiple gates would make the DRSD problem easier and not be secure. Inspired by FleXOR [84] we therefore introduce a new Free-XOR technique which, combined with our use of splitter gates, allows garbling XOR gates for free without additional assumptions. For each arbitrary fan-in XOR gate g, each party chooses a different offset Δ_g^i, allowing for a Free-XOR computation for wires using keys with that offset. For general circuits, this would normally introduce the problem that the input wires may not have the correct offset, requiring some 'translation' to Δ_g. However, because we restrict to gates with fan-out-one and splitter gates, we know that each input wire to g is not an input wire to any other gate, so we can always ensure the keys use the correct offset without any further changes.

5.4.1.3 Compiling to fan-out-1 circuits with splitter gates

Let C_f be an arbitrary fan-out circuit, with A AND gates and X XOR gates, both with fan-in-two. Let I_{C_f} and O_{C_f} be the number of circuit-input and circuit-output wires, respectively. We will now compute the number S of splitter gates that the compiled circuit needs. First, note that each time a wire w is used as input to another gate or as a circuit-output wire, w's fan-out is increased by one. Each of the AND, XOR gates in the pre-compiled circuit provides a fresh output wire to be used in C_f, while using for its inputs two pre-existing wires in the circuit. Output wires also

Functionality $\mathcal{F}_{\text{Bit}\times\text{Bit}}$

After receiving $(x^i, y^i) \in \mathbb{F}_2^2$ from each party P_i, sample $z_i \leftarrow \mathbb{F}_2$ such that $\sum_i z^i = (\sum_i x^i) \cdot (\sum_i y^i)$, and send z^i to party P_i.

Figure 5.8: Secret-shared bit multiplication functionality.

Functionality $\mathcal{F}^{\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}(P_j)$

After receiving $(x^i, y^i) \in \mathbb{F}_2^2$ from each party P_i, as well as $\Delta \in \mathbb{F}_2^{\ell_{\text{BMR}}}$ from P_j, sample $Z_i \leftarrow \mathbb{F}_2$ such that $\sum_i Z^i = (\sum_i x^i) \cdot \Delta$, and send Z^i to party P_i.

Figure 5.9: Secret-shared bit/string multiplication functionality.

use one pre-existing wire each, while input wires use no pre-existing wires. This means that, to compile C_f to be a fan-out-one circuit, we need to add up to $(2 \cdot X + 2 \cdot A + O_{C_f}) - (A + X + I_{C_f})$ wires. Each of these missing wires, however, can be created by using a splitter gate in the compiled circuit, since each of these gates uses one wire to generate two fresh new wires. So, putting all the pieces together, the compiled circuit requires $S \leq X + A + O_{C_f} - I_{C_f}$ splitter gates. This gives a close upper bound, as if w is a circuit output wire *and* an input wire of another gate then it is being counted twice rather than once in the formula.

5.4.2 Protocol and Functionalities for Bit and Bit/String Multiplication

Even though we could implement both $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ and $\mathcal{F}^{\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}(P_j)$ using $\mathcal{F}_{\text{Triple}}$, there are more efficient ways to implement the latter: One by building directly from $\mathcal{F}_{\text{Leaky-2-Mult}}$, and another using [10].

- $\mathcal{F}_{\text{Leaky-2-Mult}}$-hybrid implementation (Figure 5.11): As the length-ℓ_{BMR} string R_g^j is not secret-shared and just known to one party, we only need to perform $n-1$ invocations of $\mathcal{F}_{\text{Leaky-2-Mult}}$ in order to multiply it with a secret-shared bit $x = x^1 + \cdots + x^n$. The protocol uses random shares of zero to mask the inputs and outputs of $\mathcal{F}_{\text{Leaky-2-Mult}}$, similarly to the Π_{Triple} protocol.

  Note that this does not directly implement the functionality shown in Figure 5.9, because $\Pi^{r,\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$ performs a batch of r independent multiplications in parallel. However, in the protocol $\Pi^{\ell_{\text{BMR}}}_{\text{Preprocessing}}$ all the gates can be garbled in parallel, so a batch version of the functionality (as described in Figure 5.10) suffices. The amortized communication complexity obtained is $\ell_{\text{BMR}}(1 + \ell_{\text{OT}} + \ell_{\text{OT}}\kappa/r)$ bits.

- Asharov et al. [10] implementation: The amortized communication complexity is $\kappa + \ell_{\text{BMR}}$ bits.

Functionality $\mathcal{F}^{r,\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$

After receiving input $(P_j, x^i_1, \ldots, x^i_m)$ from every party P_i, and additional inputs $\Delta_1, \ldots, \Delta_r$ from P_j, where each $x^i_k \in \{0,1\}$ and $\Delta_k \in \{0,1\}^{\ell_{\text{BMR}}}$:

1. Sample $Z^i_k \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$, for $i \in [n]$ and $k \in [r]$, subject to the constraint that

$$\bigoplus_i Z^i_k = \Delta_k \cdot \bigoplus_i x^i_k, \quad \text{for } k \in [r]$$

2. Output $Z^i_1, \ldots, Z^i_r$ to party P_i

Figure 5.10: Batch secret-shared bit/string multiplication between P_j and all parties.

Protocol $\Pi^{r,\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$, n-party Bit/String-Mult

To multiply the strings $\Delta_1, \ldots, \Delta_r \in \{0,1\}^{\ell_{\text{BMR}}}$ held by P_j with secret-shared bits $(x^i_1, \ldots, x^i_r)_{i\in[n]}$:

1. Denote the v-th bit of Δ_k by $\Delta_{k,v}$. For $v \in [\ell_{\text{BMR}}]$:
   a) Call $\mathcal{F}^{2r}_{\text{Zero}}$ so that each P_i obtains fresh shares $(\rho^i_{1,v}, \ldots, \rho^i_{m,v}, \sigma^i_{1,v}, \ldots, \sigma^i_{m,v})$, such that $\bigoplus_i \rho^i_{k,v} = 0$ and $\bigoplus_i \sigma^i_{k,v} = 0$
   b) For each $i \neq j$, P_i and P_j run $\mathcal{F}^{r,\ell_{\text{OT}}}_{\text{Leaky-2-Mult}}$ on input $(x^i_k \oplus \sigma^i_{k,v})_{k\in[r]}$ from P_i and $(\Delta_k[v])_{k\in[r]}$ from P_j. P_i receives $a^i_{k,v}$ and P_j receives $b^i_{k,v}$ such that $a^i_{k,v} \oplus b^i_{k,v} = \Delta_k[v] \cdot (x^i_k \oplus \sigma^i_k)$.
2. Each P_i, for $i \neq j$, outputs the ℓ_{BMR}-bit strings $Z^i_k := (a^i_{k,1} \oplus \rho^i_{k,1}, \ldots, a^i_{k,\ell_{\text{BMR}}} \oplus \rho^i_{k,\ell_{\text{BMR}}})$, for $k \in [r]$.
3. P_j outputs the ℓ_{BMR}-bit strings $Z^j_k := \bigoplus_{i\neq j}(b^i_{k,1}, \ldots, b^i_{k,\ell_{\text{BMR}}}) \oplus (\rho^j_{k,1}, \ldots, \rho^j_{k,\ell_{\text{BMR}}})$, for $k \in [r]$.

Figure 5.11: n-party secret-shared bit/string multiplication using leaky 2-party multiplication.

5.4.2.1 Communication complexity and security

The communication complexity of $\Pi^{r,\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$ is exactly that of $(n-1)\ell_{\text{BMR}}$ instances of $\mathcal{F}^{r,\ell_{\text{OT}}}_{\text{Leaky-2-Mult}}$, where ℓ_{OT} is the leakage parameter used in the protocol $\Pi^{r,\ell_{\text{OT}}}_{\text{Leaky-2-Mult}}$. Note that ℓ_{OT} is independent of ℓ_{BMR} used in the bit/string protocol, but affects the security and cost of realising $\mathcal{F}_{\text{Leaky-2-Mult}}$. The total complexity is then $(n-1)\ell_{\text{BMR}}(\ell_{\text{OT}}(r+\kappa)+r)$ bits, or an amortized cost of $(n-1)\ell_{\text{BMR}}(\ell_{\text{OT}} + \ell_{\text{OT}}\kappa/r + 1)$ bits per multiplication.

Theorem 5.4.1. *Protocol $\Pi^{r,\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$ UC-securely realizes $\mathcal{F}^{r,\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$ in the $\mathcal{F}^{2r}_{\text{Zero}}$-hybrid in the presence of static honest-but-curious adversaries, under the $\mathsf{DRSD}_{r,h,\ell_{\text{OT}}}$ assumption.*

The proof is a direct extension of the proof of Theorem 5.3.2.

5.4.3 The Preprocessing Protocol

Our protocol for generating the garbled circuit is shown in Figure 5.12. We use two functionalities $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ (Figure 5.8) and $\mathcal{F}_{\text{Bit}\times\text{String}}(P_j)$ (Figure 5.9) for multiplying two additively shared bits, and multiplying an additively shared bit with a string held by P_j, respectively. $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ can be

easily implemented using a multiplication triple from $\mathcal{F}_{\text{Triple}}$ in the previous section, whilst $\mathcal{F}_{\text{Bit}\times\text{String}}$ uses a variant of the Π_{Triple} protocol optimized for this task.

The Preprocessing Protocol – $\Pi^{\ell_{\text{BMR}}}_{\text{Preprocessing}}$

COMMON INPUT: $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^{\ell_{\text{BMR}}} \to \{0,1\}^{n\ell_{\text{BMR}}+1}$, a uniformly random sampled function and H' defined from H excluding the least significant bit of the output. A boolean circuit C_f with fan-out 1. Let $\mathsf{AND}, \mathsf{XOR}$ and SPLIT be the sets of AND, XOR and splitter gates, respectively. Given a gate, let I and O be the set of its input and output wires, respectively. If $g \in \mathsf{SPLIT}$, then $I = \{w\}$ and $O = \{u, v\}$, otherwise $O = \{w\}$.
For each $i \in [n]$, the protocol proceeds as follows:

1. **Free-XOR offsets:** For every $g \in \mathsf{XOR}$, P_i samples a random value $\Delta^i_g \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$
2. **Circuit-input wires' masks and keys:** If w is a *circuit-input* wire:
   a) P_i samples a key $k^i_{w,0} \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$ and a wire mask share $\lambda^i_w \leftarrow \{0,1\}$.
   b) If w is input to a XOR gate g', P_i sets $k^i_{w,1} = k^i_{w,0} \oplus \Delta^i_{g'}$, otherwise $k^i_{w,1} \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$.
3. **Intermediate wires' masks and keys:** Passing topologically through all the gates $g \in G = \{\mathsf{AND} \cup \mathsf{XOR} \cup \mathsf{SPLIT}\}$ of the circuit:
   a) If $g \in \mathsf{XOR}$, P_i computes:
      - $\lambda^i_w = \bigoplus_{x \in I} \lambda^i_x$.
      - $k^i_{w,0} = \bigoplus_{x \in I} k^i_{x,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus \Delta^i_g$.
   b) If $g \notin \mathsf{XOR}$, P_i does as follows:
      - If $g \in \mathsf{AND}$, $\lambda^i_w \leftarrow \{0,1\}$. Else if $g \in \mathsf{SPLIT}$, sets $\lambda^i_x = \lambda^i_w$ for every $x \in O$.
      - For every $x \in O$, $k^i_{x,0} \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$. If $x \in O$ is input to a XOR gate g', set $k^i_{x,1} = k^i_{x,0} \oplus \Delta^i_{g'}$, otherwise sample $k^i_{x,1} \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$.
4. **Garble gates:** For each gate $g \in \{\mathsf{AND} \cup \mathsf{SPLIT}\}$, the parties run the subprotocol $\Pi^{\ell_{\text{BMR}}}_{\text{GateGarbling}}$, obtaining back shares $\tilde{g}^i$ of each garbled gate.
5. **Reveal input/output wires' masks:** For every *circuit-output* wire w, P_i broadcasts λ^i_w. For every *circuit-input* wire w, P_i sends λ^i_w to the party P_j who provides input on it. Each party reconstructs the wire masks from her received values as $\lambda_w = \bigoplus_{i=1}^n \lambda^i_w$.
6. **Open Garbling** For each $g \in \{\mathsf{AND} \cup \mathsf{SPLIT}\}$, P_i sends $\tilde{g}^i$ to P_1. P_1 reconstructs every garbled gate, $\tilde{g} = \bigoplus_{i=1}^n \tilde{g}^i$, and broadcasts it.

Figure 5.12: The preprocessing protocol that realizes $\mathcal{F}_{\text{Preprocessing}}$.

Most of the preprocessing protocol is similar to previous works [25, 70], where first each party samples their sets of wire keys and shares of wire masks, and then the parties interact to obtain shares of the garbled gates. It is the second stage where our protocol differs, so we focus here on the details of the gate garbling procedures.

The Gate Garbling sub-protocol – $\Pi^{\ell_{\text{BMR}}}_{\text{GateGarbling}}$

COMMON INPUT: a function $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^{\ell_{\text{BMR}}} \to \{0,1\}^{n\ell_{\text{BMR}}+1}$; H' defined as H excluding the least significant output bit; the gate g to be garbled.
PRIVATE INPUT: Each P_i, $i \in [n]$, holds λ^i_v and $k^i_{v,0}, k^i_{v,1}$, for each wire v.

1. If $g \in \mathsf{AND}$ with input wires $\{u,v\}$ and output wire w:
   a) Each party P_i defines $R^i_g = k^i_{w,0} \oplus k^i_{w,1}$, for each $i \in [n]$
   b) Call $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ to compute shares of $\lambda_u \cdot \lambda_v$, and use these to locally obtain shares of
   $$\chi_{g,a,b} = (a \oplus \lambda_u) \cdot (b \oplus \lambda_v) \oplus \lambda_w, \quad \text{for } (a,b) \in \{0,1\}^2$$
   c) Call $\mathscr{F}^{\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}(P_i)$ to get shares of $\chi_{g,a,b} \cdot R^i_g$, for each $i \in [n]$ and $(a,b) \in \{0,1\}^2$. P_i then sets $\rho^i_{i,a,b} = k^i_{w,0} \oplus (\chi_{g,a,b} \cdot R^i_g)^i$, and $\forall j \neq i$, P_j sets $\rho^j_{i,a,b} = (\chi_{g,a,b} \cdot R^i_g)^j$.
   d) Each P_i sets $\tilde{g}^i_{a,b} = \mathsf{H}(i,b,k^i_{u,a}) \oplus \mathsf{H}(i,a,k^i_{v,b}) \oplus (\chi^i_{g,a,b}, \rho^i_{1,a,b}, \ldots, \rho^i_{n,a,b})$, for $a,b \in \{0,1\}$.
2. If $g \in \mathsf{SPLIT}$ with input wire w and output wires $\{u,v\}$:
   a) Call $\mathscr{F}^{2n\ell_{\text{BMR}}}_{\text{Zero}}$ twice, so that each P_i receives shares $s^i_0, s^i_1 \in \{0,1\}^{2n\ell_{\text{BMR}}}$.
   b) P_i sets $\rho^i_c = s^i_c \oplus (0,\ldots,k^i_{u,c},0,\ldots,k^i_{v,c},\ldots,0)$ for $c \in \{0,1\}$.
   c) Set $\tilde{g}^i_c = \left(\mathsf{H}'(i,0,k^i_{w,c}), \mathsf{H}'(i,1,k^i_{w,c})\right) \oplus \rho^i_c$, for $c \in \{0,1\}$.

Figure 5.13: The gate garbling sub-protocol.

5.4.3.1 The Gate Garbling Protocol

We describe the details of the sub-protocol $\Pi^{\ell_{\text{BMR}}}_{\text{GateGarbling}}$ (Figure 5.13), implementing the gate garbling phase of $\mathscr{F}_{\text{Preprocessing}}$. Creating garbled AND gates is done similarly to the OT-based protocol [25], with the exception that we use short wire keys of length ℓ_{BMR} instead of κ. We also show how to create sharings of garbled splitter gates *without any interaction*, so these are much cheaper than AND gates.

Suppose that for an AND gate g, each P_i holds the wire mask share λ^i_v and keys $k^i_{v,0}, k^i_{v,1} \leftarrow \{0,1\}^{\ell_{\text{BMR}}}$. P_i defines $R^i_g = k^i_{w,0} \oplus k^i_{w,1}$. After that all parties call $\mathscr{F}_{\text{Bit}\times\text{Bit}}$ once to compute additive shares of $\lambda_{uv} = \lambda_u \cdot \lambda_v \in \{0,1\}$, which are then used to locally compute shares of $\chi_{g,a,b} = (a \oplus \lambda_u) \cdot (b \oplus \lambda_v) \oplus \lambda_w$, for each $(a,b) \in \{0,1\}^2$. Each P_i obtains $\chi^i_{g,a,b}$ such that $\chi_{g,a,b} = \oplus_{i\in[n]} \chi^i_{g,a,b}$. To compute shares of the products $\chi_{g,a,b} \cdot R^i_g$, the parties call $\mathscr{F}^{\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}(P_i)$ three times, for each $i \in [n]$, to multiply R^i_g with each of the bits $\lambda_u, \lambda_v, (\lambda_{uv} \oplus \lambda_w)$. These can then be used for each P_j to locally obtain the shares $(\chi_{g,a,b} \cdot R^i_g)^j$, for all $(a,b) \in \{0,1\}^2$ (just as in [25]).

After computing the bit/string products, P_j then computes for each $(a,b) \in \{0,1\}^2$:

$$\rho^j_{i,a,b} = \begin{cases} (\chi_{g,a,b} \cdot R^i_g)^j & j \neq i \\ k^i_{w,0} \oplus (\chi_{g,a,b} \cdot R^i_g)^i & j = i. \end{cases}$$

These values define shares of $\chi_{g,a,b} \cdot R^i_g \oplus k^i_{w,0}$. Finally, each party's share of the garbled AND

gate is obtained as:

$$\tilde{g}^i_{a,b} = \mathsf{H}(i,b,k^i_{u,a}) \oplus \mathsf{H}(i,a,k^i_{v,b}) \oplus (\chi^i_{g,a,b}, \rho^i_{1,a,b}, \ldots, \rho^i_{n,a,b}), \quad a,b \in \{0,1\}$$

Summing up these values we obtain:

$$\begin{aligned}\bigoplus_i \tilde{g}^i_{a,b} &= \bigoplus_i \mathsf{H}(i,b,k^i_{u,a}) \oplus \mathsf{H}(i,a,k^i_{v,b}) \oplus (\chi^i_{g,a,b}, \rho^i_{1,a,b}, \ldots, \rho^i_{n,a,b}) \\ &= \bigoplus_{i=1}^{n} (\mathsf{H}(i,b,k^i_{u,a}) \oplus \mathsf{H}(i,a,k^i_{v,b})) \oplus (c, k^1_{w,c}, \ldots, k^n_{w,c}),\end{aligned}$$

where $c = \chi_{g,a,b}$, as required.

To garble a splitter gate, we observe that here there is no need for any secure multiplications within MPC, and the parties can produce shares of the garbled gate *without any interaction*. This is because the two output wire values are the same as the input wire value, so to obtain a share of the encryption of the two output keys on wires u,v with input wire w, party P_i just computes:

$$(\mathsf{H}'(i,0,k^i_{w,c}), \mathsf{H}'(i,1,k^i_{w,c})) \oplus (0,\ldots,k^i_{u,c},0,\ldots,k^i_{v,c},0,\ldots,0)$$

for $c \in \{0,1\}$, where the right-hand vector contains P_i's keys in positions i and $n+i$. The parties then re-randomize this sharing with a share of zero from $\mathcal{F}_{\text{Zero}}$, so that opening the shares does not leak information on the individual keys.[4]

5.4.4 Complexity and Security

The above approach reduces size of the garbled circuit by a factor κ/ℓ_{BMR}, for ℓ_{BMR}-bit keys, but still requires n keys for every row in the garbled gates. Similarly to Section 5.3, when n is large we can reduce this by using a (random) committee $\mathcal{P}_{(h)}$ of size n_h that has at least h honest parties. $\Pi^{\ell_{\text{BMR}}}_{\text{Preprocessing}}$ and $\Pi^{\ell_{\text{BMR}}}_{\text{BMR}}$ are then run as if called only by the parties in $\mathcal{P}_{(h)}$. For circuit-input wires w where parties in $\mathcal{P} \setminus \mathcal{P}_{(h)}$ provide input, they are sent the masks λ_w in $\Pi^{\ell_{\text{BMR}}}_{\text{Preprocessing}}$, so in $\Pi^{\ell_{\text{BMR}}}_{\text{BMR}}$ (Figure 5.14) they can then broadcast $\Lambda_w = \rho^i_w \oplus \lambda_w$ in the same way as parties in $\mathcal{P}_{(h)}$.

This reduces the size of the garbled circuit by an additional factor of n/n_h. Finally, the same committee $\mathcal{P}_{(h)}$ can be combined with a (random) committee $\mathcal{P}_{(1)}$ with a single honest party in order to optimize the bit multiplications needed to compute the $\chi_{g,a,b}$ values, as was described in Section 5.3.

In Section 5.5, we give some examples of committee sizes and key lengths that ensure security, and compare this with the naive approach of running the preprocessing phase of BMR in $\mathcal{P}_{(1)}$ only.

[4] For AND gates, the shares output by $\mathcal{F}^{\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}$ are uniformly random, so do not need re-randomizing with sharings of zero.

Theorem 5.4.2. *Protocol* $\Pi^{\ell_{\text{BMR}}}_{\text{Preprocessing}}$ *UC-securely realizes functionality* $\mathcal{F}_{\text{Preprocessing}}$ *(Figure 5.7) with perfect security in the* $(\mathcal{F}_{\text{Bit}\times\text{Bit}}, \mathcal{F}^{\ell_{\text{BMR}}}_{\text{Bit}\times\text{String}}, \mathcal{F}^{2n\ell_{\text{BMR}}}_{\text{Zero}})$*-hybrid model in the presence of static honest-but-curious adversaries.*

Proof. Let $\mathcal{A}$ denote a PPT adversary corrupting a subset of parties $A \subset [n]$, then we prove that there exists a PPT simulator $\mathcal{S}$ that simulates the adversary's view. In the following, we denote by $\bar{A}$ the set of honest parties. When we say that the simulator is given some value, we mean that it receives it from $\mathcal{F}_{\text{Preprocessing}}$.

DESCRIPTION OF THE SIMULATION: Denote by W and O_{C_f}, respectively, the set of wires and the set of circuit-output wires of a boolean circuit C_f. Denote by $I_{C_f,S}$ the set of circuit-input wires of a circuit where a subset of parties $S \subset [n]$ provides input to the circuit. We assume w.l.o.g. that $\mathcal{A}$ is a deterministic adversary, which receives as additional input a random tape that determines its internal coin tosses. Upon receiving $\mathcal{A}$'s input $(1^\kappa, A, C_f)$ and output $(\{\lambda_w\}_{w\in O_{C_f}}, \{k^i_{v,0}, k^i_{v,1}\}_{v\in W}, \{\lambda_u\}_{u\in I_{C_f,A}})$, $\mathcal{S}$ incorporates $\mathcal{A}$ and internally emulates an execution of the honest parties running $\Pi^{\ell_{\text{BMR}}}_{\text{Preprocessing}}$ with the adversary $\mathcal{A}$.

1. CIRCUIT-INPUT WIRES' MASKS AND KEYS: For every *circuit-input* wire u and for $j \in A$, $\mathcal{S}$ samples from P_j's random tape the wire mask shares λ^j_u and the keys $k^j_{u,0}, k^j_{u,1}$ that party is meant to obtain from $\mathcal{F}_{\text{Preprocessing}}$. If a corrupted P_j provides input to the circuit on a given wire u, $\mathcal{S}$ samples $\{\lambda^i_u\}_{i\notin A}$ such that $\bigoplus_{i\notin A} \lambda^i_u = \lambda_u \oplus \bigoplus_{j\in A} \lambda^j_u$, where the value λ_u was received from $\mathcal{F}_{\text{Preprocessing}}$. If it is a honest party providing input on u, $\mathcal{S}$ samples $\{\lambda^i_u\}_{i\notin A}$ uniformly at random.

2. INTERMEDIATE WIRES' MASKS AND KEYS: Passing topologically through the gates g of the circuit:

   - For $j \in A$: If $g \in \mathsf{AND}$, $\mathcal{S}$ samples $\lambda^j_w \in \{0,1\}$ from P_j's random tape. If $g \in \mathsf{SPLIT}$, it sets $\lambda^j_x = \lambda^j_w$ for both output wires $x = u, v$. If $g \in \mathsf{XOR}$, it sets $\lambda^j_w = \bigoplus_{x\in I} \lambda^j_x$.
   - For $i \notin A$: If $g \in \mathsf{AND}$, $\mathcal{S}$ samples λ^i_w. If $g \in \mathsf{SPLIT}$, it sets $\lambda^i_x = \lambda^i_w$ for both output wires $x = u, v$. If $g \in \mathsf{XOR}$, it sets $\lambda^i_w = \bigoplus_{x\in I} \lambda^i_x$.
   - If x is a *circuit-output* wire, the simulator adjusts the value $\lambda_x \in \{0,1\}$ that $\mathcal{F}_{\text{Preprocessing}}$ sends to the parties to be $\lambda_x = \bigoplus_{i\notin A} \lambda^i_x \oplus \bigoplus_{j\in A} \lambda^j_x$.

3. GARBLE GATES: For each $g \in \mathsf{AND} \cup \mathsf{SPLIT}$:

   - If $g \in \mathsf{AND}$, let u_g, v_g be its input wires and w_g its output wire. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Bit}\times\text{Bit}}$ by sampling shares z^j_g from P_j's random tape, for $j \in A$, and setting random z^i_g for $i \notin A$ such that $\sum_{i\in[n]} z^i_g = \lambda_{u_g} \cdot \lambda_{v_g}$, where $\lambda_{u_g}, \lambda_{v_g}$ were obtained from $\mathcal{F}_{\text{Preprocessing}}$. $\mathcal{S}$ has now all the values to compute shares of $\chi_{g,a,b}$ as $\chi^i_{g,a,b} = a \cdot b \oplus b \cdot \lambda^i_{u_g} \oplus a \cdot \lambda^i_{v_g} \oplus z^i_g \oplus \lambda^i_{w_g}$ for $i \in [n]$.

For $j \in [n]$, $\mathcal{S}$ emulates three calls to $\mathcal{F}_{\text{Bit}\times\text{String}}^{\ell_{\text{BMR}}}(P_j)$ with inputs $\{\chi^i_{g,0,0}, \chi^i_{g,0,1}, \chi^i_{g,1,0}\}$ from every P_i and additional input R^j_g from P_j, where $R^j_g = k^j_{w_g,0} \oplus k^j_{w_g,1}$. In each of these emulated calls and for $i \in A$, it computes the corrupted parties' output shares from P_i's random tape, while for $i \notin A$ it samples random shares that sum to each of the values $R^j_g \cdot \chi_{g,0,0}$, $R^j_g \cdot \chi_{g,0,1}$ and $R^j_g \cdot \chi_{g,1,0}$ as required.

- If $g \in \mathsf{SPLIT}$, $\mathcal{S}$ emulates twice $\mathcal{F}_{\text{Zero}}^{2n\ell_{\text{BMR}}}$ by computing shares s^i_0, s^i_1 from P_i's random tape for $i \in A$ and setting s^j_0, s^j_1 for $j \notin A$ such that $\bigoplus_{i\in[n]} s^i_c = 0$ for $c \in \{0,1\}$.

Setting the ρ and $\tilde{g}$ values is local computation.

4. REVEAL INPUT/OUTPUT WIRES' MASKS: For every *circuit-output* wire w, $\mathcal{S}$ adds values λ^i_w, $i \notin A$ (previously computed in Step 2) to the view of each P_j, $j \in A$. For every *circuit-input* wire u on which a P_j, $j \in A$, provides input, $\mathcal{S}$ adds the $\{\lambda^i_u\}_{i\notin A}$ values it previously computed in Step 1 to P_j's view.

5. OPEN GARBLING: Using the adversary's output $\{\tilde{g}\}_{g\in\mathsf{AND}\cup\mathsf{SPLIT}}$, $\mathcal{S}$ proceeds as follows: If $1 \in A$, it plays the role of each P_j, for $j \notin A$, and sends to P_1 the shares $\{\tilde{g}^j\}_{g\in\mathsf{AND}\cup\mathsf{SPLIT}}$ that it previously computed. Otherwise, the simulator plays the role of P_1 by sending $\{\tilde{g}\}_{g\in\mathsf{AND}\cup\mathsf{SPLIT}}$ to each $P_i, i \in A$.

INDISTINGUISHABILITY: The wire keys and the (circuit-input and circuit-output) wire masks output by the functionality $\mathcal{F}_{\text{Preprocessing}}$ are i.i.d. uniformly random variables in the real world too. In both worlds and for the additional simulated values, the corrupted parties' shares for the wire masks, the bit products ($\mathcal{F}_{\text{Bit}\times\text{Bit}}$ functionality) and bit/string products ($\mathcal{F}_{\text{Bit}\times\text{String}}^{\ell_{\text{BMR}}}$ functionality) needed to garble AND gates are fixed by $\mathcal{A}$'s random tape, while the honest parties' shares of the same values are uniformly random additive shares. In particular, this implies that shares $\tilde{g}^i_{a,b}$ of garbled AND gates are uniformly random additive shares in both executions. The same applies to shares of garbled splitter gates, due to the use of the $\mathcal{F}_{\text{Zero}}^{2n\ell_{\text{BMR}}}$ functionality in the real world. Regarding the OPEN GARBLING step, if $1 \notin A$ the reconstructed garbled circuit is identically distributed in both worlds. Else, if $1 \in A$, the adversary gets additive shares of the garbled circuit both in the real and simulated executions, as we argued.

Finally, the distribution of the variables corresponding to additive shares, on the one hand, and that of the i.i.d. variables, on the other hand, guarantees that the joint output of all parties, together with the simulated/real view of corrupted parties, are identically distributed in both worlds. More formally, let $\texttt{output}^\pi(\boldsymbol{x},\kappa)$ (resp. $f(\boldsymbol{x})$) be the output of $\Pi_{\text{Preprocessing}}^{\ell_{\text{BMR}}}$ (resp. $\mathcal{F}_{\text{Preprocessing}}$) on input $\boldsymbol{x} \in \{0,1\}^*$ from all parties and security parameter κ. Let $\texttt{view}^\pi_A(\boldsymbol{x},\kappa)$ (resp. $f_A(\boldsymbol{x})$) be the restriction of these outputs to the set of corrupted parties A. We just proved that:

$$\{(\mathcal{S}(1^\kappa, \boldsymbol{x}, f_A(\boldsymbol{x})), f(\boldsymbol{x}))\}_{\boldsymbol{x},\kappa,A} \approx \{(\texttt{view}^\pi_A(\boldsymbol{x},\kappa), \texttt{output}^\pi(\boldsymbol{x},\kappa))\}_{\boldsymbol{x},\kappa,A}.$$

■

Protocol $\Pi_{\text{BMR}}^{\ell_{\text{BMR}}}$

COMMON INPUT: A boolean circuit C_f with fan-out-one gates representing the function f. Let AND, XOR and SPLIT be the sets of AND, XOR and splitter gates, respectively. For a gate $g \in \mathsf{SPLIT}$, let the input wire be $\{w\}$ and the output wires be $\{u, v\}$. Otherwise let $\{u, v\}$ be the input wires and $\{w\}$ the output wire.
CRS: $\mathsf{H} : [n] \times \{0,1\} \times \{0,1\}^{\ell_{\text{BMR}}} \to \{0,1\}^{n\ell_{\text{BMR}}+1}$, a uniformly random function, and H' defined from H by excluding the least significant bit of the output.

The parties execute the following commands in sequence:

Preprocessing:

1. Call $\mathscr{F}_{\text{Preprocessing}}$ with input C_f. Each party P_i obtains the garbled version $\tilde{g}$ of every gate $g \in \mathsf{SPLIT} \cup \mathsf{AND}$, the wire masks λ_w for every output wire and every wire associated with their input, and all their keys $\{k^i_{w,0}, k^i_{w,1}\}$ for every wire w of the circuit.

Online Computation:

1. For all input wires w with input from P_i, party P_i computes $\Lambda_w = \rho^i_w \oplus \lambda_w$, where ρ^i_w is P_i's input to C_f on wire w, and λ_w was obtained from $\mathscr{F}_{\text{Preprocessing}}$. Then, P_i broadcasts the external value Λ_w to all parties.
2. For all input wires w, each party P_i broadcasts the key k^i_{w,Λ_w} associated to Λ_w.
3. Passing through the circuit topologically, the parties can now locally compute the following operations for each gate g.
   a) If $g \in \mathsf{SPLIT}$, set $\Lambda_x = \Lambda_w$ for $x \in \{u, v\}$ and then compute:
   $$(k^1_{u,\Lambda_u}, \ldots, k^n_{u,\Lambda_u}, k^1_{v,\Lambda_v}, \ldots, k^n_{v,\Lambda_v}) = \tilde{g}_{\Lambda_w} \oplus \left(\bigoplus_{i=1}^{n} \mathsf{H}'(i, 0, k^i_{w,\Lambda_w}), \bigoplus_{i=1}^{n} \mathsf{H}'(i, 1, k^i_{w,\Lambda_w}) \right)$$
   b) If $g \in \mathsf{AND}$, the parties compute:
   $$(\Lambda_w, k^1_{w,\Lambda_w}, \cdots, k^n_{w,\Lambda_w}) = \tilde{g}_{\Lambda_u,\Lambda_v} \oplus \bigoplus_{i=1}^{n} \left(\mathsf{H}(i, \Lambda_v, k^i_{u,\Lambda_u}) \oplus \mathsf{H}(i, \Lambda_u, k^i_{v,\Lambda_v}) \right)$$
   c) If $g \in \mathsf{XOR}$, the parties compute $\Lambda_w = \bigoplus_{x \in I} \Lambda_x$ and $k^i_{w,\Lambda_w} = \bigoplus_{x \in I} k^i_{x,\Lambda_x}$ for $i \in [n]$.
4. Eventually, all parties will obtain an external value Λ_w for every circuit-output wire w. Each party can then recover the actual output value from $\rho_w = \Lambda_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

Figure 5.14: Online phase of the constant-round MPC protocol.

5.4.5 The Online Phase

We present the online phase of our protocol for multi-party garbled circuits with short keys in Figure 5.14. Given the previous description of the garbling phase, the online phase is quite straightforward, where upon reconstructing the garbled circuit and obtaining all input keys, the evaluation process is similar to that of [18], described in Section 2.6. As in that work, all parties run the evaluation algorithm, which in our case involves each party computing just

$2n$ hash evaluations per gate. As usual, during the evaluation parties obtain external values corresponding to a random mask of the actual values being computed. Upon completion, the parties obtain the actual output using the output wire masks revealed from $\mathscr{F}_{\text{Preprocessing}}$. The security of the protocol reduces to the $\mathsf{DRSD}_{r,h,\ell_{\text{BMR}}}$ problem, where ℓ_{BMR} is the key length, h is the number of honest parties, and r is twice the output length of the function H (sampled by the CRS).

We remark that in practice, we may want to implement the random function H in the CRS using fixed-key AES in the ideal cipher model, as is common for garbling schemes based on Free-XOR. In Section 5.5.2, we show that this reduces the number of AES calls from $O(n^2)$ in previous BMR protocols to $O(n^2\ell_{\text{BMR}}/\kappa)$.

We conclude with the following theorem

Theorem 5.4.3. *Let f be an n-party functionality $\{0,1\}^{n\kappa} \mapsto \{0,1\}^{\kappa}$ and assume that the $\mathsf{DRSD}_{2r,h,\ell_{\text{BMR}}}$ assumption (see Definition 5.5) holds, where $r = n\ell_{\text{BMR}} + 1$. Then Protocol $\Pi^{\ell_{\text{BMR}}}_{\text{BMR}}$ from Figure 5.14 UC-securely computes f in the presence of a static honest-but-curious adversary corrupting $t = n-h$ parties in the $\mathscr{F}_{\text{Preprocessing}}$-hybrid model.*

Proof. We reduce security of the protocol to the extended double-key decisional-RSD problem (Definition 5.7) with parameters (r,h,ℓ), where $r := n\ell_{\text{BMR}} + 1$. By Lemma 5.2.5, this is reducible to $\mathsf{DRSD}_{2r,h,\ell}$.

Let $\mathscr{A}$ be a PPT adversary corrupting a subset of parties $A \subset [n]$ such that $|A| = n-h$. We prove that there exists a PPT simulator $\mathscr{S}$, with access to an ideal functionality $\mathscr{F}$ that implements f, which simulates the adversary's view. The simulator fixes the CRS as a random $2n \cdot 2^{\ell_{\text{BMR}}} \times 2^{n\ell_{\text{BMR}}+1}$ matrix. A key k_w for wire w is denoted as an active key if it is observed by the adversary upon evaluating the garbled circuit. The remaining hidden key is denoted as an inactive key. An active path is the set of all active keys that are observed throughout the garbled circuit evaluation.

Denoting the set of honest parties by $\bar{A}$, our simulator $\mathscr{S}$ is defined below.

THE DESCRIPTION OF THE SIMULATION:

1. INITIALIZATION. Upon receiving the adversary's input $(1^\kappa, A, \boldsymbol{x}_A)$ and output $\boldsymbol{y}$, $\mathscr{S}$ samples a i.i.d uniformly random tapes r_i for each $i \in A$, incorporates $\mathscr{A}$ and internally emulates an execution of the honest parties running $\Pi^{\ell_{\text{BMR}}}_{\text{BMR}}$ with the adversary $\mathscr{A}$. When we say that $\mathscr{S}$ chooses a value for some corrupted party, we mean that it samples the value from that party's random tape r_i.

2. PREPROCESSING. $\mathscr{S}$ obtains the adversary's input C_f which is a Boolean circuit that computes f with a set of wires W and a set of G gates, and emulates $\mathscr{F}_{\text{Preprocessing}}$, as follows:

   - For every XOR gate g and $i \in A$ the simulator samples $\Delta^i_g \in \{0,1\}^{\ell_{\text{BMR}}}$.

- For every input wire u the simulator chooses a random bit $\Lambda_u \in \{0,1\}$ and, for every $i \in \bar{A}$, an active key $k^i_{u,\Lambda_u} \in \{0,1\}^{\ell_{\text{BMR}}}$. Additionally, it chooses a key $k^i_{u,0} \in \{0,1\}^{\ell_{\text{BMR}}}$ for every $i \in A$. Finally, and also for $i \in A$, if u is input to a XOR gate g' it sets $k^i_{u,1} = k^i_{u,0} \oplus \Delta^i_{g'}$, otherwise it samples $k^i_{u,1} \in \{0,1\}^{\ell_{\text{BMR}}}$.

The simulator continues the emulation of the garbling phase by computing an active path of the garbled circuit that corresponds to the sequence of keys which will be observed by the adversary. Importantly, $\mathscr{S}$ never samples the inactive keys $k^i_{u,\bar{\Lambda}_u}, k^i_{v,\bar{\Lambda}_v}$ and $k^i_{w,\bar{\Lambda}_w}$ for $i \in \bar{A}$ in order to generate the garbled circuit.

- ACTIVE PATH GENERATION OF XOR GATES. For every XOR gate g with input a set of wires I and an output wire w,
  - $\mathscr{S}$ sets $\Lambda_w = \bigoplus_{x \in I} \Lambda_x$.
  - Next, for $i \in A$ it sets $k^i_{w,0} = \bigoplus_{x \in I} k^i_{x,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus \Delta^i_g$.
  - Finally, for $i \in \bar{A}$ the simulator sets $k^i_{w,\Lambda_w} = \bigoplus_{x \in I} k^i_{x,\Lambda_x}$.
- ACTIVE PATH GENERATION OF AND GATES. For every AND gate g with input wires $I = \{u,v\}$ and an output wire w, $\mathscr{S}$ samples a random $\Lambda_w \in \{0,1\}$ and honestly generates the entry in row (Λ_u, Λ_v), where Λ_u (resp. Λ_v) is the external value associated to the left (resp. right) input wire to g. Namely, the simulator computes

$$\tilde{g}_{\Lambda_u,\Lambda_v} = \left(\bigoplus_{i=1}^{n} \mathsf{H}(i, \Lambda_v, k^i_{u,\Lambda_u}) \oplus \mathsf{H}(i, \Lambda_u, k^i_{v,\Lambda_v}) \right) \oplus (\Lambda_w, k^1_{w,\Lambda_w}, \ldots, k^n_{w,\Lambda_w}).$$

  The remaining three rows are sampled uniformly at random from $\{0,1\}^{n\ell_{\text{BMR}}+1}$.

- ACTIVE PATH GENERATION OF SPLITTER GATES. For every splitter gate g with an input wire $I = \{w\}$ and output wires $O = \{u,v\}$, $\mathscr{S}$ sets $\Lambda_x = \Lambda_w$ for every $x \in O$ and honestly generates the entry in row Λ_w, where Λ_w is the external value associated to the input wire to g. Namely, the simulator computes

$$\tilde{g}_{\Lambda_w} = \left(\bigoplus_{i=1}^{n} \mathsf{H}'(i, 0, k^i_{w,\Lambda_w}), \bigoplus_{i=1}^{n} \mathsf{H}'(i, 1, k^i_{w,\Lambda_w}) \right) \oplus (k^1_{u,\Lambda_u}, \ldots, k^n_{u,\Lambda_u}, k^1_{v,\Lambda_v}, \ldots, k^n_{v,\Lambda_v}).$$

  The remaining row is sampled uniformly at random from $\{0,1\}^{2n\ell_{\text{BMR}}}$.

- SETTING THE TRANSLATION TABLE. For every output wire $w \in W$ returning the ith bit of $\boldsymbol{y}$, the simulator sets $\lambda_w = \Lambda_w \oplus y_i$. For all input wires $w \in W''$ that are associated with the ith bit of $\boldsymbol{x}_A$ (the adversary's input), the simulator sets $\lambda_w = \Lambda_w \oplus \boldsymbol{x}_{A,i}$. The simulator forwards the adversary the λ_w value for every output wire $w \in W$ and every circuit-input wire $w \in W''$ associated with a corrupted party. It completes the emulation of $\mathscr{F}_{\text{Preprocessing}}$ by adding the complete garbled circuit to the view of each corrupted party.

3. ONLINE COMPUTATION. In the online computation the simulator adds to the view of every corrupted party the external values $\{\Lambda_w\}_{w\in W'}$ that are associated with the honest parties' input wires W'. The simulator adds the honest parties' input keys $\{k^i_{w,\Lambda_w}\}_{i\in\bar{A},w\in W'}$ to the view of each corrupted party.

This concludes the description of the simulation. Note that the difference between the simulated and the real executions is regarding the way the garbled circuit is generated. More concretely, the simulated garbled gates include a single row that is properly produced, whereas the remaining three rows are picked at random. Let $\mathbf{HYB}^{\mathcal{F}_{\text{Preprocessing}}}_{\Pi^{\ell_{\text{BMR}}}_{\text{BMR}},\mathcal{A},\mathcal{Z}}(1^\kappa,z)$ denote the output distribution of the adversary $\mathcal{A}$ and honest parties in a real execution using $\Pi^{\ell_{\text{BMR}}}_{\text{BMR}}$ with adversary $\mathcal{A}$. Moreover, let $\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa,z)$ denote the output distribution of $\mathcal{S}$ and the honest parties in an ideal execution.

We prove that the ideal and real executions are indistinguishable.

Lemma 5.4.1. *The following two distributions are computationally indistinguishable:*

- $\{\mathbf{HYB}^{\mathcal{F}_{\text{Preprocessing}}}_{\Pi^{\ell_{\text{BMR}}}_{\text{BMR}},\mathcal{A},\mathcal{Z}}(1^\kappa,z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$
- $\{\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa,z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$

Proof: We begin by defining a slightly modified simulated execution $\widetilde{\mathbf{HYB}}$, where the generation of the garbled circuit is modified so that upon receiving the parties' inputs $\{\rho^i\}_{i\in[n]}$ the simulator $\widetilde{\mathcal{S}}$ first evaluates the circuit C_f, computing the actual bit ρ_w to be transferred via wire w for all $w\in W$, where W is the set of wires of C_f. It then chooses wire mask shares and wire keys as in the description of functionality $\mathcal{F}_{\text{Preprocessing}}$ from Figure 5.7. Finally, $\widetilde{\mathcal{S}}$ fixes the active key for each wire $w\in W$ to be $(k^1_{w,\rho_w\oplus\lambda_w},\ldots,k^n_{w,\rho_w\oplus\lambda_w})$. The rest of this hybrid is identical to the simulation. This hybrid execution is needed in order to construct a distinguisher for the Extended Double-Key RSD assumption.
Let $\widetilde{\mathbf{HYB}}^{\mathcal{F}_{\text{Preprocessing}}}_{\Pi^{\ell_{\text{BMR}}}_{\text{BMR}},\mathcal{A}}(1^\kappa,z)$ denote the output distribution of the adversary $\mathcal{A}$ and honest parties in this game. It is simple to verify that the adversary's views in $\widetilde{\mathbf{HYB}}$ and $\mathbf{IDEAL}$ are identical, as in both cases the garbling of each gate includes just a single row that is correctly garbled and the external value associated with each wire w is independent of $\ell_{\text{BMR}w}$.

Our proof of the lemma follows by a reduction to the Extended Double-Key RSD hardness assumption (cf. Definition 5.7). Assume by contradiction the existence of an environment $\mathcal{Z}$, an adversary $\mathcal{A}$ and a non-negligible function $p(\cdot)$ such that

$$\left|\Pr[\mathcal{Z}(\mathbf{HYB}^{\mathcal{F}_{\text{Preprocessing}}}_{\Pi^{\ell_{\text{BMR}}}_{\text{BMR}},\mathcal{A},\mathcal{Z}}(1^\kappa,z))=1]-\Pr[\mathcal{Z}(\widetilde{\mathbf{HYB}}_{\Pi^{\ell_{\text{BMR}}}_{\text{BMR}},\mathcal{A},\mathcal{Z}}(1^\kappa,z))=1]\right|\geq\frac{1}{p(\kappa)}$$

for infinitely many κ's where the probability is taken over the randomness of $\mathcal{Z}$ as well as the randomness for choosing the Λ values and the keys. Then we construct a PPT distinguisher $\mathcal{D}$ for

the Extended Double-Key RSD assumption that distinguishes between an instance of the form

$$\left(\mathsf{H}, \bigoplus_{i\in\bar{A}} \mathsf{H}(i,0,k_i), \bigoplus_{i\in\bar{A}} \mathsf{H}(i,0,k'_i), \bigoplus_{i\in\bar{A}} \mathsf{H}(i,1,k_i), \bigoplus_{i\in\bar{A}} \mathsf{H}(i,1,k'_i)\right)$$

and five random elements, for some subset $\bar{A}$ of $[n]$ of size h (that corresponds to the set of honest parties) with probability at least $\frac{1}{p(\kappa)\cdot|C|}$ via a sequence of hybrid games $\{\mathbf{HYB}_i\}_{i\in[|C|]}$, where $C = \mathsf{SPLIT} \cup \mathsf{AND}$. In more details, we define hybrid $\mathbf{HYB}_i$ as a hybrid execution with a simulator $\mathscr{S}_i$ that garbles the circuit as follows. The first i gates in the topological order are garbled as in the simulation whereas the remaining $|C| - i$ gates are garbled as in the real execution. Note that $\mathbf{HYB}_0$ is distributed as hybrid $\mathbf{HYB}$ and that $\mathbf{HYB}_{|C|}$ is distributed as $\widetilde{\mathbf{HYB}}$. Therefore, if $\mathbf{HYB}$ and $\widetilde{\mathbf{HYB}}$ are distinguishable with probability $\frac{1}{p(\kappa)}$ then there exists $\tau \in [|C|]$ such that hybrids $\mathbf{HYB}_{\tau-1}$ and $\mathbf{HYB}_\tau$ are distinguishable with probability at least $\frac{1}{p(\kappa)\cdot|C|}$. Next, we formally describe our reduction to the Extended Double-Key RSD hardness assumption. Upon receiving a tuple $(\mathsf{H}, \widetilde{\mathsf{H}}_0, \widetilde{\mathsf{H}}'_0, \widetilde{\mathsf{H}}_1, \widetilde{\mathsf{H}}'_1)$ that is distributed according to the first or the second distribution, a subset $\bar{A}$ of $[n]$ that denotes the set of honest parties, an index τ and the environment's input z, distinguisher $\mathscr{D}$ internally invokes $\mathscr{Z}$ and simulator $\mathscr{S}$. In more detail:

- $\mathscr{D}$ internally invokes $\mathscr{Z}$ that fixes the honest parties' inputs $\boldsymbol{\rho}$.
- $\mathscr{D}$ emulates the communication with the adversary (controlled by $\mathscr{Z}$) in the initialization, preprocessing and garbling steps as in the simulation with $\mathscr{S}$.
- For each wire u, let $\rho_u \in \{0,1\}$ be the actual value on wire u. Note that these values, as well as the output of the computation y, can be determined since $\mathscr{D}$ knows the actual input of all parties to the circuit.
- For each wire u in the circuit and $i \in A$, $\mathscr{D}$ chooses a pair of keys $k^i_{u,0}, k^i_{u,1} \in \{0,1\}^{\ell_{\mathrm{BMR}}}$, whereas for all $i \in \bar{A}$ it samples a random key $k^i_{u,\Lambda_u} \in \{0,1\}^{\ell_{\mathrm{BMR}}}$. $\mathscr{D}$ further fixes the external value $\Lambda_u = \lambda_u \oplus \rho_u$.
- $\mathscr{D}$ then garbles the circuit as follows.
  - For every $g_j \in \mathsf{AND}$ with input wires u and v and output wire w, $\mathscr{D}$ continues as follows.

    If $j < \tau$ then $\mathscr{D}$ garbles g_j exactly as in the simulation with $\widetilde{\mathscr{F}}$.

    If $j = \tau$ then $\mathscr{D}$ first honestly computes the (Λ_u, Λ_v)-th row by fixing

    $$\tilde{g}_{\Lambda_u,\Lambda_v} = \left(\bigoplus_{i=1}^{n} \mathsf{H}(i, \Lambda_v, k^i_{u,\Lambda_u}) \oplus \mathsf{H}(i, \Lambda_u, k^i_{v,\Lambda_v})\right) \oplus (c, k^1_{w,\Lambda_w}, \ldots, k^n_{w,\Lambda_w})$$

    where $c = \Lambda_w$.

    Next, $\mathscr{D}$ samples an inactive key $k^i_{w,\bar{\Lambda}_w}$ for all $i \in \bar{A}$ and fixes the remaining three rows as follows.

$$\tilde{g}_{\Lambda_u,\bar{\Lambda}_v} = \left(\bigoplus_{i=1}^{n} \mathsf{H}(i,\bar{\Lambda}_v,k^i_{u,\Lambda_u}) \oplus \left(\bigoplus_{i\in A} \mathsf{H}(i,\Lambda_u,k^i_{v,\bar{\Lambda}_v})\right) \oplus \widetilde{\mathsf{H}}'_{\Lambda_u}\right)$$
$$\oplus (c,k^1_{w,c},\ldots,k^n_{w,c}), \quad \text{where } c = \Lambda_u\cdot\bar{\Lambda}_v \oplus \Lambda_w \oplus \rho_w$$
$$\tilde{g}_{\bar{\Lambda}_u,\Lambda_v} = \left(\bigoplus_{i\in A} \mathsf{H}(i,\Lambda_v,k^i_{u,\bar{\Lambda}_u}) \oplus \widetilde{\mathsf{H}}_{\Lambda_v} \oplus \left(\bigoplus_{i=1}^{n} \mathsf{H}(i,\bar{\Lambda}_u,k^i_{v,\Lambda_v})\right)\right)$$
$$\oplus (c,k^1_{w,c},\ldots,k^n_{w,c}), \quad \text{where } c = \bar{\Lambda}_u\cdot\Lambda_v \oplus \Lambda_w \oplus \rho_w$$
$$\tilde{g}_{\bar{\Lambda}_u,\bar{\Lambda}_v} = \left(\bigoplus_{i\in A} \mathsf{H}(i,\bar{\Lambda}_v,k^i_{u,\bar{\Lambda}_u}) \oplus \widetilde{\mathsf{H}}_{\bar{\Lambda}_v} \oplus \left(\bigoplus_{i\in A} \mathsf{H}(i,\bar{\Lambda}_u,k^i_{v,\bar{\Lambda}_v})\right) \oplus \widetilde{\mathsf{H}}'_{\bar{\Lambda}_u}\right)$$
$$\oplus (c,k^1_{w,c},\ldots,k^n_{w,c}), \quad \text{where } c = \bar{\Lambda}_u\cdot\bar{\Lambda}_v \oplus \Lambda_w \oplus \rho_w.$$

Finally, if $j > \tau$ then $\mathscr{D}$ garbles g_j exactly as in hybrid **HYB**. For that, $\mathscr{D}$ needs to know both active and inactive keys. It therefore chooses the inactive keys that are associated with the input and output wires of this gate for $i \in \bar{A}$, in order to be able to complete the garbling. Recall that the circuit is with fan-out 1. Therefore the distinguisher can choose the inactive key for the input wire of this gate (as it was not used as an input wire to gate g_τ).

– For every $g_j \in \mathsf{SPLIT}$ with input wire w and output wires u, v, $\mathscr{D}$ completes the garbling as follows.

If $j < \tau$ then $\mathscr{D}$ garbles g_j exactly as in the simulation with $\widetilde{\mathscr{S}}$.

If $j = \tau$ then $\mathscr{D}$ first honestly computes the Λ_wth row by fixing

$$\tilde{g}_{\Lambda_w} = \left(\bigoplus_{i=1}^{n} \mathsf{H}(i,0,k^i_{w,\Lambda_w}) \oplus \mathsf{H}(i,1,k^i_{w,\Lambda_w})\right) \oplus (k^1_{u,\Lambda_u},\ldots,k^n_{v,\Lambda_v}).$$

Next, it samples inactive keys $k^i_{u,\bar{\Lambda}_u}, k^i_{v,\bar{\Lambda}_v}$ for all $i \in \bar{A}$ and fixes the remaining row as follows.

$$\tilde{g}_{\bar{\Lambda}_w} = \left(\bigoplus_{i\in A} \mathsf{H}(i,0,k^i_{w,\bar{\Lambda}_w}) \oplus \widetilde{\mathsf{H}}_0 \oplus \bigoplus_{i\in A} \mathsf{H}(i,1,k^i_{w,\bar{\Lambda}_w}) \oplus \widetilde{\mathsf{H}}_1\right) \oplus (k^1_{u,\bar{\Lambda}_u},\ldots,k^n_{v,\bar{\Lambda}_v}).$$

If $j > \tau$ then $\mathscr{D}$ garbles g_j as in hybrid **HYB** using a similar process as for the case of an AND gate.

- This concludes the description of the reduction. Note that the set XOR need not be part of these hybrids since we do not send any garbling information for this set of gates. $\mathscr{D}$ hands the adversary the complete description of the garbled circuit and concludes the execution as in the simulation with $\widetilde{\mathscr{S}}$.

- $\mathscr{D}$ outputs whatever $\mathscr{Z}$ does.

Note first that if $(\widetilde{\mathsf{H}}_0, \widetilde{\mathsf{H}}'_0, \widetilde{\mathsf{H}}_1, \widetilde{\mathsf{H}}'_1)$ are truly uniform then the view generated by $\mathscr{D}$ is distributed as in $\mathbf{HYB}_\tau$. This is because only the active path is created as in the real execution, whereas the remaining rows are sampled uniformly at random from the appropriate domain. On the other hand, if this tuple is generated according to the following distribution

$$\left(\mathsf{H}, \bigoplus_{i\in A}\mathsf{H}(i,0,k_i), \bigoplus_{i\in A}\mathsf{H}(i,0,k'_i), \bigoplus_{i\in A}\mathsf{H}(i,1,k_i), \bigoplus_{i\in A}\mathsf{H}(i,1,k'_i)\right)$$

then this emulates game $\mathbf{HYB}_{\tau-1}$, since each tuple element emulates an evaluation of the hash values for the honest parties on the secret keys.

□

This concludes the proof of Theorem 5.4.3. ■

5.5 Complexity Analysis and Implementation Results

We now compare the complexity of the most relevant aspects of our approach to the state-of-the-art prior results in semi-honest MPC protocols with dishonest majority. To demonstrate the practicality of our approach, we also present implementation results for the online evaluation phase of our BMR-based protocol.

5.5.1 Threshold Variants of Full-Threshold Protocols

Since the standard GMW and BMR-based protocols allow for up to $n-1$ corruptions, we also show how to modify previous protocols to support some threshold t, and compare our protocols with these variants. The method is very simple (and similar to the use of committees in our protocols), but does not seem to have been explicitly mentioned in previous literature. To evaluate a circuit C, all parties first secret-share their inputs to an arbitrarily chosen committee $\mathscr{P}'$, of size $t+1$. Committee $\mathscr{P}'$ runs the full-threshold protocol for a modified circuit C', which takes all the shares as input, and first XORs them together so that it computes the same function as C. The committee $\mathscr{P}'$ then sends the output to all parties in $\mathscr{P}$. The complexity of the threshold-t variant of a full-threshold protocol, Π, is then essentially the same as running Π between $t+1$ parties instead of n.

5.5.2 Instantiating the CRS

Our protocols require a CRS, which is a randomly sampled function, H. One way of implementing this would be generate the function in a setup phase (e.g. with coin-tossing) and store it as a lookup table. However, when the table grows large this will have a prohibitive impact on performance, as there will likely be many cache misses when reading from H at random locations. A more efficient alternative is to implement H using fixed-key AES, which offers fast performance

on modern CPUs with AES hardware instructions. This gives security in the ideal cipher model, where fixed-key AES is modelled as a random permutation.[5]

Depending on which of our two protocols is used, this method works as follows:

- For GMW, H is a 1-bit output hash function, so we can simply truncate the AES output.
- For our BMR-style protocol, we need to expand the input to $n \cdot \ell + 1$ bits. Let $B = \lceil (n \cdot \ell + 1)/128 \rceil$ be the number of AES blocks needed to generate one hash output. The parties first fix a random key $s \leftarrow \{0,1\}^{128}$ and then define:

$$\mathsf{H}(i,b,k) = (\mathsf{AES}_s(i \| b \| k \| 0), \ldots, \mathsf{AES}_s(i \| b \| k \| B-1)),$$

  where the last block is truncated so that the total output length is $n \cdot \ell_{\text{BMR}} + 1$ bits.

  The cost of a single call to H is that of B AES operations.

5.5.3 Concrete Hardness of RSD and Our Choice of Parameters

In this section we give an overview of how we select the key length ℓ in our protocols according to n, h, r, so that the corresponding $\mathsf{RSD}_{r,h,\ell}$ instance is hard enough. For a more detailed survey of known attacks and the techniques involved, check the full version [69] of the paper this chapter is based on. As discussed in Section 5.2.4, RSD is similar to the (standard) syndrome decoding problem, where each component of the error vector is 0 or 1 with some constant probability, which is equivalent to the problem of *learning parity with noise* (LPN).

The most efficient attacks on RSD are Information Set Decoding (ISD), introduced by Prange in 1962 [112], Wagner's Generalised Birthday Attack (GBA) [127], and the Linearization Attack (LA) by Bellare et al. [22] and Saarinen [120]. We stress that the goal of our analysis is to find a reasonable estimation of the complexity of these attacks; giving a complete description of all possible decoding techniques and a precise evaluation of their cost is out of the scope of this thesis. In our analysis we intentionally underestimate the complexity of all the attacks, resulting in a conservative estimate of the security of our protocols.

When considering the hardness of RSD instances we need to distinguish the case where the solution to the problem is unique and the case of multiple solutions. In the first case, which always occurs for our BMR-style protocols, GBA essentially reduces to the classical birthday attack and the most efficient attack is ISD. Classical information set decoding algorithms do not take into account the possibility that the solution is regular. In practice, when we estimate the cost of this attack, we consider the cost of both a tailored regular variant of ISD, augmented with the Stern [125] and Finiasz and Sendrier [56] techniques, and the more recent non-regular variant due to

[5] This actually only provides security up to the birthday bound, i.e. as long as the adversary makes no more than 2^{64} queries to AES_s. In practice, however, since ℓ is small there will typically be far fewer than 2^{64} possible inputs, so we do not need to be concerned with the birthday bound.

n	20			50				80				100				200		400	
h	10	11	16	10	20	25	40	16	24	32	56	20	30	40	60	60	100	80	120
ℓ Pra	19	18	13	21	13	11	8	32	12	10	8	14	11	9	8	8	8	8	8
ℓ	32	29	18	>32	27	16	8	>32	30	17	8	>32	25	15	8	14	8	11	8

Table 5.1: Min key-length for BMR-style MPC with 128 bits of security for different n and h when $r = 2\ell n + 2$

h	15		20		30		40		50		80	
r	300	1500	300	3000	300	2000	400	3000	450	1000	420	2500
ℓ	14	26	11	32	8	16	7	15	6	8	4	8

Table 5.2: Min key-length for GMW-style MPC with 128 bits of security for different n and h

# parties n (honest)	20 (6)	50 (15)	60 (20)	80 (30)	150 (40)	200 (50)	400 (120)
(ℓ_{OT}, r)	(31,300)	(14,300)	(11,300)	(8,300)	(7,400)	(6,450)	(1,80)
GMW ($t = n-1$)	25.46	164.15	237.18	423.44	1497.5	2666.6	10693.2
GMW ($t = n-h$)	14.07	84.42	109.88	170.85	818.07	1517.55	5271.56
Ours	**12.89**	**37**	**40.38**	**50.01**	**169.36**	**261.6**	**403.63**

Table 5.3: Amortized communication cost (in kbit) of producing a single triple in GMW. We consider [51] for 1-out-of-4 OT extension in the GMW protocols, and the protocol from Section 5.3 in our work.

Becker et al. [19], and then we take the minimum of the two. We have also analysed more recent variants of ISD [32, 98], see [69] for more details.

In Table 5.1, we provide an estimation of the minimal key-length ℓ for our BMR-style protocols to achieve more than 128 bits of security for different values of n, h and $r = 2\ell n + 2$. Note that we only consider $8 \leq \ell \leq 32$, so when in the table we have that ℓ should be larger than 32, it means that ISD cost less than 2^{128} for that set of parameters. We also give an estimation of minimal key-length respect to the plain ISD attack to RSD by Prange.

When an RSD instance has more than one solution – this is sometimes the case for our GMW-style protocol – we need to consider also GBA and LA. Notice that since there are many solutions, attacking regular SD with classical ISD is more difficult than attacking non-regular SD and an adversary needs to run the attack repeatedly until the output is regular, increasing the cost of the attack. To estimate the complexity of GBA and LA we take the same conservative approach we use for ISD. Since LA is particularly effective for larger h, especially when $h > r/4$, we always set up $r > 2h + 1$.

In Table 5.2 we propose a set of of parameters for our GMW-style protocols for different values of h (and irrespective to the total number of parties n), such that the estimated complexity of the most efficient decoding algorithms is larger than 2^{128}.

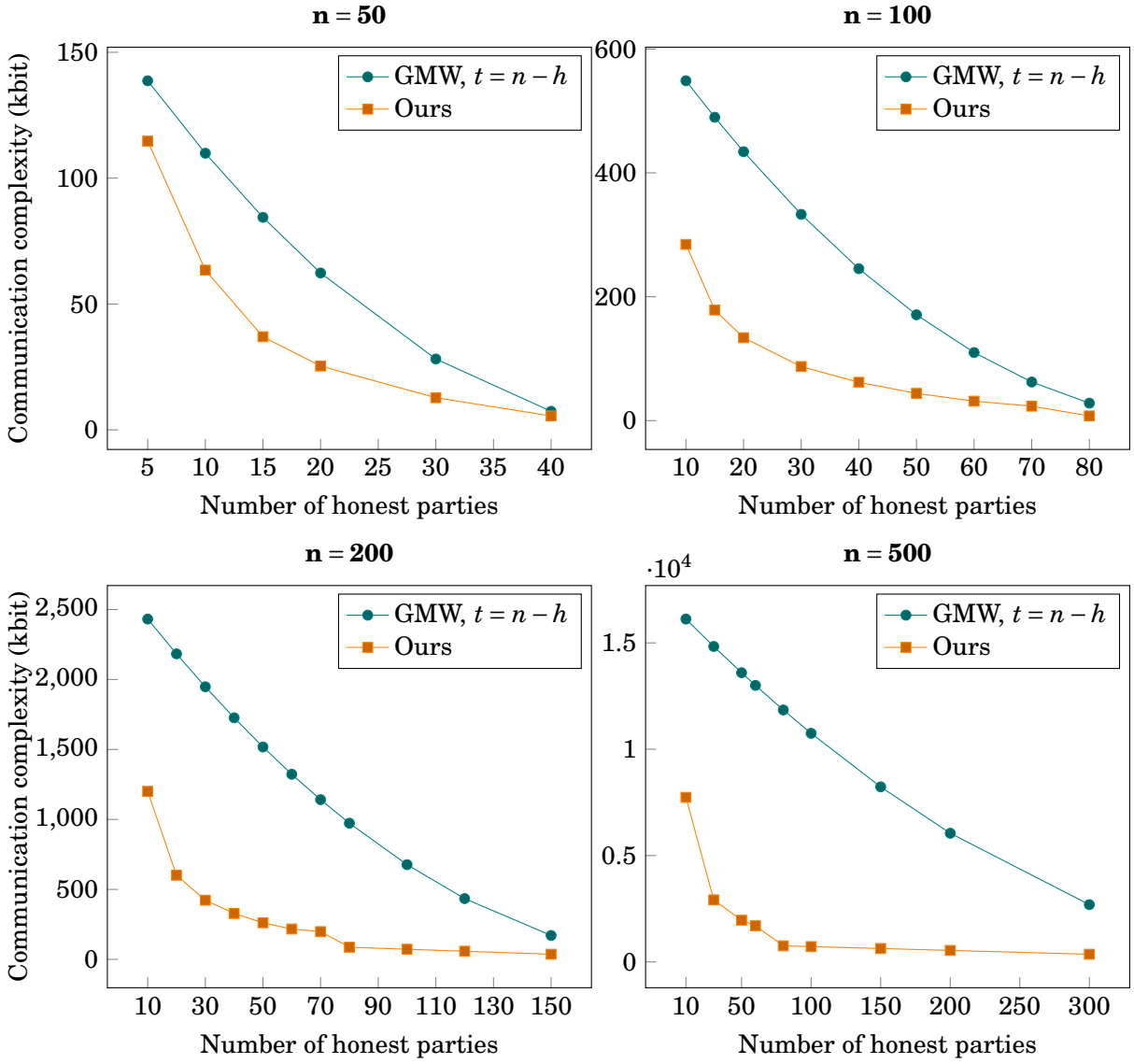


Figure 5.15: Amortized communication cost (in kbit) for producing triples in GMW for $n = 50, 100, 200, 500$ and deterministic committees.

5.5.4 GMW-Style Protocol

We now compare the communication cost of our triple generation protocol with the best-known instantiation of GMW, namely a variant based on 1-out-of-4 OT to generate triples, recently optimized by [51] in the 2-party setting. This easily extends to the multi-party case with communication complexity $O(n^2\kappa/\log\kappa)$ bits per AND gate, so we consider both full-threshold and threshold-t (Section 5.5.1) variants. Note that our protocol from Section 5.3 has complexity $O(nt\ell)$ when using deterministic committees, with ℓ as in Table 5.1.

As can be seen in Table 5.3 and Figure 5.15, for a fixed number of honest parties h, the improvement of our protocol over GMW (threshold t) becomes greater as the total number of parties increases. Our protocol starts to beat the best-known GMW protocol for producing multiplication triples when there are just 6 honest parties. For example, with 20 parties and 14 corruptions, the communication cost of our protocol is roughly 10% lower than threshold-14 GMW, and only 2 times lower than the cost of standard, full threshold GMW. As the number of parties (and honest parties) grows, our improvements become even greater, and when the number of honest parties is more than 80, we can use 1-bit keys and improve upon the threshold variant of GMW by *more than* 13 *times*.

In Section 5.3 we mentioned the possibility, when n and h are large enough, of using *random committees* $\mathscr{P}_{(h)}$ and $\mathscr{P}_{(1)}$, such that except with negligible probability $\mathscr{P}_{(h)}$ has at least $h' \leq h$ honest parties and $\mathscr{P}_{(1)}$ has at least one honest party. In order to estimate the communication complexity of our protocol, we consider the probability $p_{(1)}$ of $\mathscr{P}_{(1)}$ not having a single honest party and the probability $p_{(h)}$ of $\mathscr{P}_{(h)}$ of having less than h' honest parties. Let $n_1 = |\mathscr{P}_{(1)}|$ and $n_h = |\mathscr{P}_{(h)}|$, we have that

$$p_{(1)} = \frac{\binom{n-h}{n_1}}{\binom{n}{n_1}} \quad \text{and} \quad p_{(h)} = \frac{\sum_{j=1}^{\min(h',h'-v)} \binom{n-h}{n_h-h'+j} \cdot \binom{h}{h'-j}}{\binom{n}{n_h}},$$

where $v = n_h - (n-h) < h'$. We fix the parameters n, h, h', and compute the minimum values n_h, n_1 such that $p_{(h)}$ and $p_{(1)}$ are less than 2^{-s}. Table 5.4 compares our protocol with random committees and GMW with a single random committee of size n_1, i.e. having at least one honest party with overwhelming probability, when $s = 40$. Even if the communication complexity reduces in both protocols, our approach is always at least 50% more efficient compared to GMW.

(n,h,h')	(100, 40, 30)	(200, 70, 50)	(500, 200, 120)	(800, 300, 120)	(1000, 200, 120)	(5000,1200,120)	(10000, 3000, 120)
(ℓ_{OT},r)	(8, 300)	(6, 450)	(1, 80)	(1, 80)	(1, 80)	(1, 80)	(1, 80)
$(n_{h'},n_1)$	(90, 39)	(180, 54)	(382, 51)	(447, 57)	(790, 117)	(811, 100)	(654, 78)
GMW	99.3	191.75	170.85	213.9	909.32	663.3	402.40
Ours	**43.6**	**84.62**	**70.13**	**91.72**	**337.75**	**291**	**183.64**

Table 5.4: Amortized communication cost (in kbit) of producing a single triple in GMW using random committees.

# parties (honest)	20 (10)	50 (20)	80 (32)	100 (40)	200 (60)	400 (120)	1000 (160)
$(\ell_{\text{BMR}}, \ell_{\text{OT}}, r)$	$(32,23,530)$	$(27,13,450)$	$(17,8,380)$	$(15,7,400)$	$(8,5,370)$	$(8,1,80)$	$(8,1,120)$
[25] (Gb $\mathscr{P}$)	341.24	2200.1	5675.36	8890	35740	143320.8	897102
[25] (Gb $\mathscr{P}_{(1)}$)	**98.78**	835.14	2112.1	3286.7	17726.45	70654.7	634383.12
Ours (Garbling)	111.7	**747.63**	**1750.48**	**2678.74**	**5448.36**	**10114.99**	**64474.1**
[25] ($\|GC\|$ $\mathscr{P}$)	$10.24A$	$25.6A$	$40.96A$	$51.2A$	$102.4A$	$204.8A$	$512A$
[25] ($\|GC\|$ $\mathscr{P}_{(1)}$)	$5.632A$	$15.88A$	$25.1A$	$31.23A$	$72.19A$	$143.9A$	$430.6A$
[26] ($\|GC\|$)	$12.29(A+X)$	$12.29(A+X)$	$12.29(A+X)$	$12.29(A+X)$	$12.29(A+X)$	$12.29(A+X)$	$12.29(A+X)$
Ours ($\|GC\|$)	$2.56(A+S)$	$5.4(A+S)$	$5.45(A+S)$	$6(A+S)$	$6.4(A+S)$	12.8(A+S)	32(A+S)

Table 5.5: Communication complexity for garbling, and size of garbled gates, in BMR-style protocols in kbit. A = #AND gates, S = #Splitter gates, X = #XOR gates.

5.5.5 BMR-Style Protocol

5.5.5.1 Communication Complexity

To show the efficiency of our constant-round garbling protocol from Section 5.4.5, we provide Table 5.5, which has two parts. First, it compares the amortized communication complexity incurred for garbling an AND gate with [25]. We recall that this is the dominating cost for BMR-style protocols using Free-XOR, and that we incur no communication for creating shares of garbled splitter gates. Note that in the first setting of $n = 20, t = 10$, although our communication costs are around 3 times lower than [25], we do not improve upon the threshold-t variant of that protocol, described earlier. Once we get to 50 parties, though, we start to improve upon [25], with a reduction in communication going up to 7x for 400 parties and 10x for 1000 parties.

The second half of the table shows the size of the garbled circuit in terms of the total number of AND, XOR and splitter gates. Garbled circuit size only has a slight impact on communication complexity, when opening the garbled circuit, which is much lower than the communication in the rest of the garbling phase. However, if an implementation needs to store the entire garbled circuit in memory (either for evaluation, or storage for later use) then it is also important to optimize its size. Here we also compare with [26], which recently showed how to construct a compact multi-party garbled circuit based on key-homomorphic PRFs. The size of their garbled circuit is constant and grows with $O(\kappa)$ per gate, with security proven in the presence of $n-1$ corrupted parties. On the other hand, their construction requires much more expensive operations based on the Decisional Diffie-Hellman (DDH) or Ring-LWE assumptions, and these also lead to fairly large keys – with a 3072-bit discrete log prime (equivalent to 128-bit security) the size of a garbled AND gate only beats our protocol at around 400 parties. Additionally, their construction does not support Free-XOR, and the concrete efficiency of the offline garbling phase is not clear: garbling an AND gate requires $O(n)$ secret-shared field multiplications, which seems likely to be much higher than the offline cost of our protocol or [25], but their paper does not give concrete numbers. In Figure 5.16 we show the communication complexity of garbling when $n = 100,500$ and for different number of honest parties.

CONCRETE COST FORMULAS: Here we explicit how to obtain the communication complexity

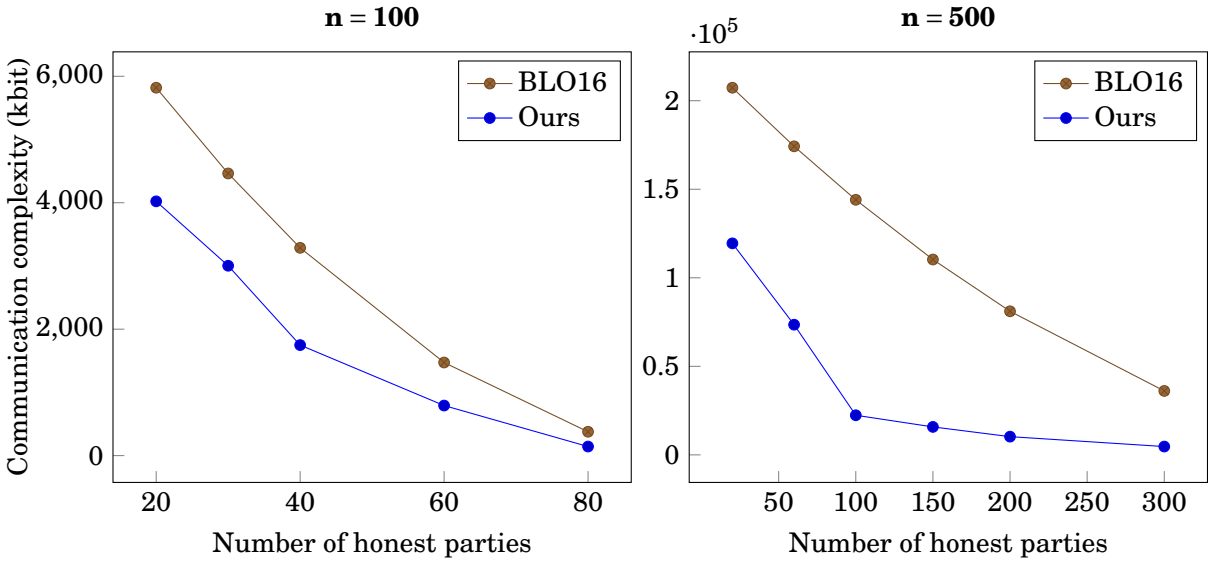


Figure 5.16: Communication complexity cost (in kbit) for garbling when $n = 100$ and $n = 500$.

results in Table 5.5 and Figure 5.16, which compare our work with the best previous passively secure BMR protocol, namely [25]. To simplify the comparison, we exclude the communication related to input and output wires. Given a circuit C_f with X XOR gates and A AND gates, each of them with fan-in-two and arbitrary fan-out, [25] has the following communication costs:

1. One bit multiplication and $3n$ bit-string multiplications per AND gate, where the strings have length κ. A bit multiplication requires $n(n-1)$ bit-OTs, each of which involves sending $128+2$ bits if instantiated with [74] or 84 bits if instantiated with [83]. Each of the bit-string multiplication can be computed using $n-1$ correlated OTs, at a cost of $128+128$ bits each.

2. Each AND gate has size $4n\kappa$ bits, and each party has a share of it. If the circuit is reconstructed by every party sending her share to P_1, and then P_1 broadcasting the addition of every share, the cost of putting an AND gate together is $8n(n-1)\kappa$ bits.

This gives a total cost of $(130+768+8\kappa)\cdot n\cdot(n-1)\cdot A = 1922\cdot n\cdot(n-1)\cdot A$ bits for the [74] instantiation and $(67+768+8\kappa)\cdot n\cdot(n-1)\cdot A = 1876\cdot n\cdot(n-1)\cdot A$ when using [51] instead. In our work, the costs are:

1. One bit multiplication and $3n$ bit/string multiplications per AND gate, where the strings have length ℓ. When implemented with our improved GMW protocol with deterministic committees, these cost $(n-h+1)(n-1)(\ell_{\text{OT}}+\ell_{\text{OT}}\kappa/r+1)$ bits and $n(n-1)\ell_{\text{BMR}}(\ell_{\text{OT}}+\ell_{\text{OT}}\kappa/r+1)$ bits, respectively.

2. Each AND gate has size $4\cdot(n\ell+1)$ bits. Each Splitter gate has size $4n\ell$ bits.

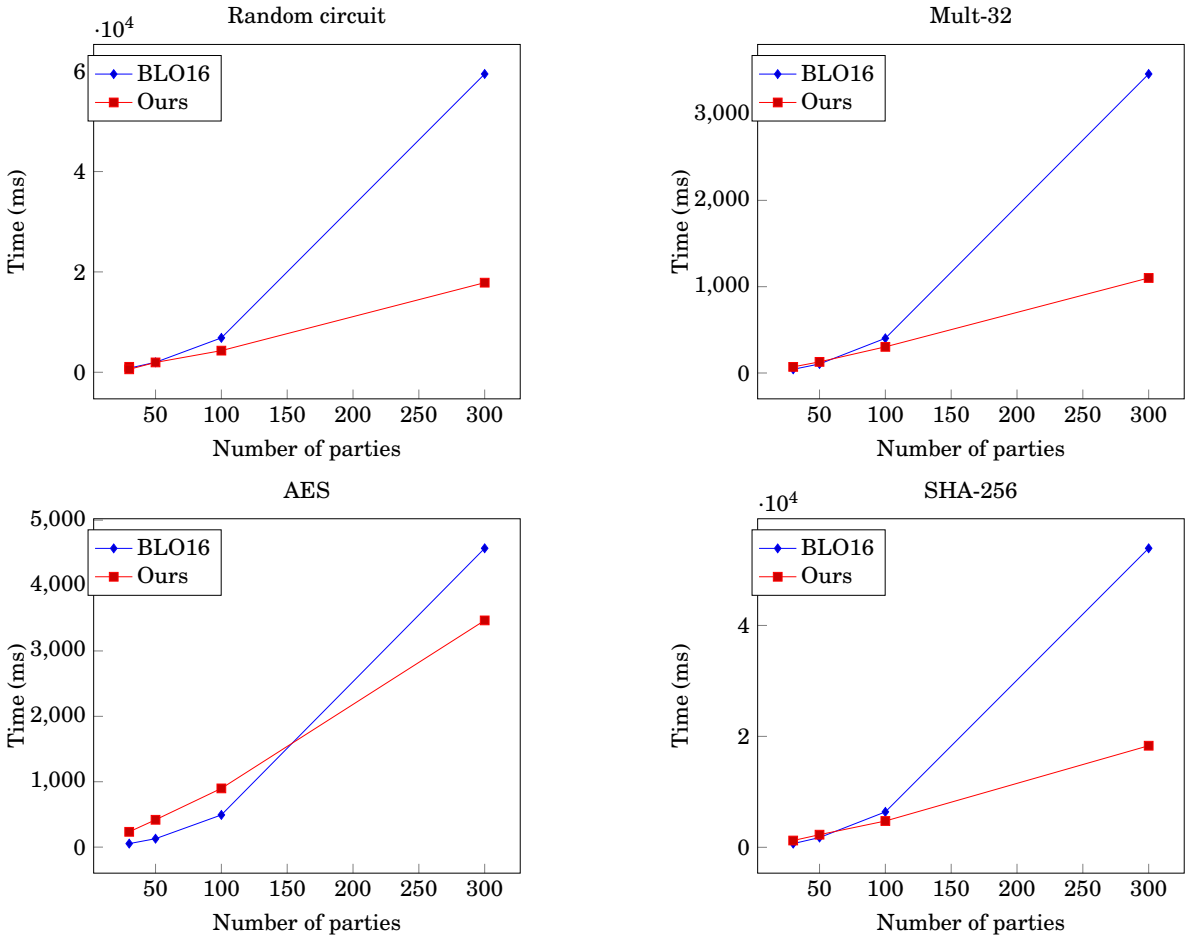


Figure 5.17: Online time for evaluating various circuits with $n = 30, 50, 100, 300$. The corresponding numbers of honest parties are $h = 14, 21, 38, 105$, respectively. Times for [25] are for a full-threshold implementation.

5.5.5.2 Garbling Implementation

Note: The following implementation work was not carried out by the author, but by Assi Barak, Moriya Farbstein and Lior Koskas from the software team at Bar-Ilan University. It appears here for completeness and a better understanding of the previous results in this chapter.

In Figure 5.17, we present running times for evaluating the garbled circuit in our protocol and compare with times for [25] running on the same machine.

The implementation runs on a single machine,[6] to allow testing just the local computation in the online phase (note that there is very little interaction in the online phase). We took as benchmarks the AES circuit (6800 AND gates, 31796 Splitters), the SHA-256 circuit (90825 AND gates, 132586 Splitters), a binary multiplier for 32-bit numbers (5926 AND gates, 6994 Splitters) and a randomly generated circuit with 100000 AND gates and 99510 Splitters (as used

[6] Intel Xeon E5-2650 v3 2.3GHz / 25M Cache, 10 Cores, 64GB RAM.

in [26], for comparison). The CRS H was implemented with fixed-key AES in counter mode using AES-NI instructions, which is a random function under the assumption that AES behaves like a random permutation (see Section 5.5.2). We also tried precomputing every output of H and storing this as a lookup-table, but in practice this did not perform well as the table size was usually much larger than the CPU cache. Recall that the standard BMR online phase requires each party to perform $O(n^2)$ AES operations per AND gate, whereas our online phase reduces this to $O(n^2\ell_{\text{BMR}}/\kappa)$, with some extra cost for evaluating splitter gates. The results show that for the random circuit our protocol starts to pay off from around 50 parties, when the corruption threshold is between 20–40%, reaching a 3.3x improvement for $n = 300, h = 105$. On the other hand, for AES, which has a relatively large proportion of splitter gates, the crossover point is closer to 150 parties, and the greatest improvement factor is 1.3x over [25] for $n = 300, h = 105$. This shows that the performance improvements of our garbled circuit-based protocol very much depend on the specific circuit being evaluated, but further improvements may be possible by modifying secure computation compilers to produce circuits more suitable for our protocol. It also seems likely that implementing our GMW-based protocol would show much more significant gains, based on the communication costs presented earlier.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

After several real-world deployments and proposals of Multi-Party Computation [29–31], this research area is now ready to scale up and explore scenarios where networks are slower or more parties take part in the computation. Whereas those might seem as differentiated directions, most probably if tens, hundreds or even thousands of parties are to perform MPC in the real world, they will have to do so through a WAN and network latency will then be an issue.

With few exceptions [26, 33], concrete efficiency for large numbers of parties does not seem to have attracted a lot of interest by the community yet. This is likely to change due to both the latest improvements and a growing number of scenarios which inherently involve data from many sources [7, 31]. With regards to these scenarios, in work not included in this thesis [68] we extended the recent TinyKeys technique presented in Chapter 5 to provide active security. This new result builds on the TinyOT family of protocols (see [58, 104, 129] or Section 4.7), for which we improve the communication complexity by producing shorter message-authentication codes. For future works based in TinyKeys, several interesting questions remain open. Perhaps most remarkably, it has to be shown how to efficiently extend BMR-style protocols with short keys, as the one described in Section 5.4, to be secure against active adversaries. Other areas of interest include adaptive adversaries and further cryptanalysis of the Regular Syndrome Decoding problem in order to improve the currently conservative parameters.

Regarding MPC over slow networks without necessarily as many parties, constant-round protocols provide a promising solution. Specially for deep boolean circuits, it has been empirically verified in the passive multi-party setting (for BMR-style protocols [25]) and in the active, two-party one [128] that garbled circuits outperform other approaches. Nevertheless, as progress on BMR-style protocols goes along, more experiments in different scenarios will be necessary in order to keep comparing with the presently more active line of research in secret-sharing based

MPC. At the moment, we know that secret-sharing based protocols win for lower-depth circuits and more than two parties when the adversary is passive [25]. Running these kind of comparisons might highlight different strengths and lead to the discovery of new, unexpected bottlenecks which are clearer in the two-party setting. Knowledge of strong and weak points might at the same time lead to hybrid secret-sharing/garbled circuits solutions where different parts of the computation are obtained using either one tool or the other. Whereas these techniques have been explored in more depth in the two-party setting [50], there is currently only one work [81] exploiting the different strengths of each approach in the multi-party setting.

On more theoretical grounds, an important step for BMR-style protocols during the length of this PhD was finding the right way to define the garbling functionality, as it can be noticed by the difference between the ones appearing in Chapters 3 and 4. Finding abstractions which both simplify the community's work when writing proofs and are general enough to cover present and future works is always an interesting task. The functionalities provided in Chapter 4 have already proved useful to other authors as explicitly mentioned in [27, 67]. Notably, those two works are of a very different nature: [27] deals with concrete efficiency improvements for BMR, while [67] constructs the first round-optimal actively secure MPC protocol under standard assumptions.

In the same vein, future BMR-based works could use help from a rigorous definition of the 'vector double encryption' concept, introduced in Chapter 2 informally and for the first time in the literature. A similar notion exists in the two party setting [91], but its extension to more parties is not straightforward even within already existing results. Such formalization would lead to a more general garbling functionality than the one provided in Chapter 4, together with a proof reducing to any 'vector double encryption' scheme instead of circular 2-correlation robust PRFs.

BIBLIOGRAPHY

[1] A. AFSHAR, P. MOHASSEL, B. PINKAS, AND B. RIVA, *Non-interactive secure computation based on cut-and-choose*, in EUROCRYPT 2014, P. Q. Nguyen and E. Oswald, eds., vol. 8441 of LNCS, Springer, Heidelberg, May 2014, pp. 387–404.

[2] G. A. AKERLOF, *The market for "lemons": Quality uncertainty and the market mechanism*, Quarterly journal of economics, 84 (1970), pp. 488–500.

[3] B. APPLEBAUM, *Garbling XOR gates"for free" in the standard model*, Journal of Cryptology, 29 (2016), pp. 552–576.

[4] B. APPLEBAUM, Y. ISHAI, AND E. KUSHILEVITZ, *Cryptography with constant input locality*, Journal of Cryptology, 22 (2009), pp. 429–469.

[5] T. AQUINAS, *Summa theologiae*, circa 1265-1274.

[6] T. ARAKI, J. FURUKAWA, Y. LINDELL, A. NOF, AND K. OHARA, *High-throughput semi-honest secure three-party computation with an honest majority*, in ACM CCS 16, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, eds., ACM Press, Oct. 2016, pp. 805–817.

[7] G. ASHAROV, D. DEMMLER, M. SCHAPIRA, T. SCHNEIDER, G. SEGEV, S. SHENKER, AND M. ZOHNER, *Privacy-preserving interdomain routing at internet scale*, PoPETs, 2017 (2017), p. 147.

[8] ——, *Privacy-preserving interdomain routing at internet scale*, PoPETs, 2017 (2017), p. 147.

[9] G. ASHAROV, A. JAIN, A. LÓPEZ-ALT, E. TROMER, V. VAIKUNTANATHAN, AND D. WICHS, *Multiparty computation with low communication, computation and interaction via threshold FHE*, in EUROCRYPT 2012, D. Pointcheval and T. Johansson, eds., vol. 7237 of LNCS, Springer, Heidelberg, Apr. 2012, pp. 483–501.

[10] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, AND M. ZOHNER, *More efficient oblivious transfer and extensions for faster secure computation*, in ACM CCS 13, A.-R. Sadeghi, V. D. Gligor, and M. Yung, eds., ACM Press, Nov. 2013, pp. 535–548.

[11] ——, *More efficient oblivious transfer extensions with security for malicious adversaries*, in EUROCRYPT 2015, Part I, E. Oswald and M. Fischlin, eds., vol. 9056 of LNCS, Springer, Heidelberg, Apr. 2015, pp. 673–701.

[12] D. AUGOT, M. FINIASZ, AND N. SENDRIER, *A fast provably secure cryptographic hash function*, IACR Cryptology ePrint Archive, 2003 (2003), p. 230.

[13] F. BACON, *Novum organum*, 1620.

[14] J. BAR-ILAN AND D. BEAVER, *Non-cryptographic fault-tolerant computing in constant number of rounds of interaction*, in 8th ACM PODC, P. Rudnicki, ed., ACM, Aug. 1989, pp. 201–209.

[15] C. BAUM, I. DAMGÅRD, T. TOFT, AND R. W. ZAKARIAS, *Better preprocessing for secure multiparty computation*, in ACNS 16, M. Manulis, A.-R. Sadeghi, and S. Schneider, eds., vol. 9696 of LNCS, Springer, Heidelberg, June 2016, pp. 327–345.

[16] D. BEAVER, *Efficient multiparty protocols using circuit randomization*, in CRYPTO'91, J. Feigenbaum, ed., vol. 576 of LNCS, Springer, Heidelberg, Aug. 1992, pp. 420–432.

[17] ——, *Correlated pseudorandomness and the complexity of private computations*, in 28th ACM STOC, ACM Press, May 1996, pp. 479–488.

[18] D. BEAVER, S. MICALI, AND P. ROGAWAY, *The round complexity of secure protocols (extended abstract)*, in 22nd ACM STOC, ACM Press, May 1990, pp. 503–513.

[19] A. BECKER, A. JOUX, A. MAY, AND A. MEURER, *Decoding random binary linear codes in $2^{n/20}$: How 1 + 1 = 0 improves information set decoding*, in EUROCRYPT 2012, D. Pointcheval and T. Johansson, eds., vol. 7237 of LNCS, Springer, Heidelberg, Apr. 2012, pp. 520–536.

[20] M. BELLARE, V. T. HOANG, S. KEELVEEDHI, AND P. ROGAWAY, *Efficient garbling from a fixed-key blockcipher*, in 2013 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, May 2013, pp. 478–492.

[21] M. BELLARE, V. T. HOANG, AND P. ROGAWAY, *Foundations of garbled circuits*, in ACM CCS 12, T. Yu, G. Danezis, and V. D. Gligor, eds., ACM Press, Oct. 2012, pp. 784–796.

[22] M. BELLARE AND D. MICCIANCIO, *A new paradigm for collision-free hashing: Incrementality at reduced cost*, in EUROCRYPT'97, W. Fumy, ed., vol. 1233 of LNCS, Springer, Heidelberg, May 1997, pp. 163–192.

[23] A. BEN-DAVID, N. NISAN, AND B. PINKAS, *FairplayMP: a system for secure multi-party computation*, in ACM CCS 08, P. Ning, P. F. Syverson, and S. Jha, eds., ACM Press, Oct. 2008, pp. 257–266.

[24] A. BEN-EFRAIM, *On multiparty garbling of arithmetic circuits*, IACR Cryptology ePrint Archive, 2017 (2017), p. 1186.

[25] A. BEN-EFRAIM, Y. LINDELL, AND E. OMRI, *Optimizing semi-honest secure multiparty computation for the internet*, in ACM CCS 16, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, eds., ACM Press, Oct. 2016, pp. 578–590.

[26] A. BEN-EFRAIM, Y. LINDELL, AND E. OMRI, *Efficient scalable constant-round MPC via garbled circuits*, in ASIACRYPT 2017, Part II, T. Takagi and T. Peyrin, eds., vol. 10625 of LNCS, Springer, Heidelberg, Dec. 2017, pp. 471–498.

[27] A. BEN-EFRAIM AND E. OMRI, *Concrete efficiency improvements for multiparty garbling with an honest majority*, in Latincrypt 2017, 2017.

[28] M. BEN-OR, S. GOLDWASSER, AND A. WIGDERSON, *Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)*, in 20th ACM STOC, ACM Press, May 1988, pp. 1–10.

[29] D. BOGDANOV, M. JÕEMETS, S. SIIM, AND M. VAHT, *How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation*, in FC 2015, R. Böhme and T. Okamoto, eds., vol. 8975 of LNCS, Springer, Heidelberg, Jan. 2015, pp. 227–234.

[30] D. BOGDANOV, L. KAMM, B. KUBO, R. REBANE, V. SOKK, AND R. TALVISTE, *Students and taxes: a privacy-preserving study using secure computation*, PoPETs, 2016 (2016), pp. 117–135.

[31] P. BOGETOFT, D. L. CHRISTENSEN, I. DAMGÅRD, M. GEISLER, T. JAKOBSEN, M. KRØIGAARD, J. D. NIELSEN, J. B. NIELSEN, K. NIELSEN, J. PAGTER, M. I. SCHWARTZBACH, AND T. TOFT, *Secure multiparty computation goes live*, in FC 2009, R. Dingledine and P. Golle, eds., vol. 5628 of LNCS, Springer, Heidelberg, Feb. 2009, pp. 325–343.

[32] L. BOTH AND A. MAY, *Decoding linear codes with high error rate and its impact for lpn security*, IACR Cryptology ePrint Archive, 2017 (2017), p. 1139.

[33] G. BRACHA, *An $O(\lg n)$ expected rounds randomized byzantine generals protocol*, in 17th ACM STOC, ACM Press, May 1985, pp. 316–326.

[34] S. S. BURRA, E. LARRAIA, J. B. NIELSEN, P. S. NORDHOLT, C. ORLANDI, E. ORSINI, P. SCHOLL, AND N. P. SMART, *High performance multi-party computation for binary circuits based on oblivious transfer*.
Cryptology ePrint Archive, Report 2015/472, 2015.
`http://eprint.iacr.org/2015/472`.

[35] R. CANETTI, *Universally composable security: A new paradigm for cryptographic protocols*, in 42nd FOCS, IEEE Computer Society Press, Oct. 2001, pp. 136–145.

[36] R. CANETTI, A. COHEN, AND Y. LINDELL, *A simpler variant of universally composable security for standard multiparty computation*, in CRYPTO 2015, Part II, R. Gennaro and M. J. B. Robshaw, eds., vol. 9216 of LNCS, Springer, Heidelberg, Aug. 2015, pp. 3–22.

[37] I. CASCUDO, I. DAMGÅRD, B. DAVID, N. DÖTTLING, AND J. B. NIELSEN, *Rate-1, linear time and additively homomorphic UC commitments*, in CRYPTO 2016, Part III, M. Robshaw and J. Katz, eds., vol. 9816 of LNCS, Springer, Heidelberg, Aug. 2016, pp. 179–207.

[38] D. CHAUM, C. CRÉPEAU, AND I. DAMGÅRD, *Multiparty unconditionally secure protocols (extended abstract)*, in 20th ACM STOC, ACM Press, May 1988, pp. 11–19.

[39] S. G. CHOI, K.-W. HWANG, J. KATZ, T. MALKIN, AND D. RUBENSTEIN, *Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces*, in CT-RSA 2012, O. Dunkelman, ed., vol. 7178 of LNCS, Springer, Heidelberg, Feb. / Mar. 2012, pp. 416–432.

[40] S. G. CHOI, J. KATZ, R. KUMARESAN, AND H.-S. ZHOU, *On the security of the "free-XOR" technique*, in TCC 2012, R. Cramer, ed., vol. 7194 of LNCS, Springer, Heidelberg, Mar. 2012, pp. 39–53.

[41] S. G. CHOI, J. KATZ, A. J. MALOZEMOFF, AND V. ZIKAS, *Efficient three-party computation from cut-and-choose*, in CRYPTO 2014, Part II, J. A. Garay and R. Gennaro, eds., vol. 8617 of LNCS, Springer, Heidelberg, Aug. 2014, pp. 513–530.

[42] A. COSTACHE AND N. P. SMART, *Which ring based somewhat homomorphic encryption scheme is best?*, in CT-RSA 2016, K. Sako, ed., vol. 9610 of LNCS, Springer, Heidelberg, Feb. / Mar. 2016, pp. 325–340.

[43] I. DAMGÅRD AND Y. ISHAI, *Scalable secure multiparty computation*, in CRYPTO 2006, C. Dwork, ed., vol. 4117 of LNCS, Springer, Heidelberg, Aug. 2006, pp. 501–520.

[44] I. DAMGÅRD, M. KELLER, E. LARRAIA, V. PASTRO, P. SCHOLL, AND N. P. SMART, *Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits*, in ESORICS 2013, J. Crampton, S. Jajodia, and K. Mayes, eds., vol. 8134 of LNCS, Springer, Heidelberg, Sept. 2013, pp. 1–18.

[45] I. DAMGÅRD AND J. B. NIELSEN, *Scalable and unconditionally secure multiparty computation*, in CRYPTO 2007, A. Menezes, ed., vol. 4622 of LNCS, Springer, Heidelberg, Aug. 2007, pp. 572–590.

[46] I. DAMGÅRD, J. B. NIELSEN, M. NIELSEN, AND S. RANELLUCCI, *The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited*, in CRYPTO 2017, Part I, J. Katz and H. Shacham, eds., vol. 10401 of LNCS, Springer, Heidelberg, Aug. 2017, pp. 167–187.

[47] I. DAMGÅRD, V. PASTRO, N. P. SMART, AND S. ZAKARIAS, *Multiparty computation from somewhat homomorphic encryption*, in CRYPTO 2012, R. Safavi-Naini and R. Canetti, eds., vol. 7417 of LNCS, Springer, Heidelberg, Aug. 2012, pp. 643–662.

[48] I. DAMGÅRD, A. POLYCHRONIADOU, AND V. RAO, *Adaptively secure multi-party computation from LWE (via equivocal FHE)*, in PKC 2016, Part II, C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, eds., vol. 9615 of LNCS, Springer, Heidelberg, Mar. 2016, pp. 208–233.

[49] I. DAMGÅRD AND S. ZAKARIAS, *Constant-overhead secure computation of Boolean circuits using preprocessing*, in TCC 2013, A. Sahai, ed., vol. 7785 of LNCS, Springer, Heidelberg, Mar. 2013, pp. 621–641.

[50] D. DEMMLER, T. SCHNEIDER, AND M. ZOHNER, *ABY - A framework for efficient mixed-protocol secure two-party computation*, in NDSS 2015, The Internet Society, Feb. 2015.

[51] G. DESSOUKY, F. KOUSHANFAR, A.-R. SADEGHI, T. SCHNEIDER, S. ZEITOUNI, AND M. ZOHNER, *Pushing the communication barrier in secure computation using lookup tables*, in Network and Distributed System Security Symposium (NDSSí17). The Internet Society, 2017.

[52] W. DIFFIE AND M. E. HELLMAN, *New directions in cryptography*, IEEE Transactions on Information Theory, 22 (1976), pp. 644–654.

[53] R. DINGLEDINE, N. MATHEWSON, AND P. F. SYVERSON, *Tor: The second-generation onion router*, in USENIX, 2004, pp. 303–320.

[54] N. M. DÖTTLING, *Cryptography based on the Hardness of Decoding*, PhD thesis, Karlsruhe Institute of Technology, 2014.

[55] *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*, Official Journal of the European Union, L119 (2016), pp. 1–88.

[56] M. FINIASZ AND N. SENDRIER, *Security bounds for the design of code-based cryptosystems*, in ASIACRYPT 2009, M. Matsui, ed., vol. 5912 of LNCS, Springer, Heidelberg, Dec. 2009, pp. 88–105.

[57] T. K. FREDERIKSEN, T. P. JAKOBSEN, AND J. B. NIELSEN, *Faster maliciously secure two-party computation using the GPU*, in SCN 14, M. Abdalla and R. D. Prisco, eds., vol. 8642 of LNCS, Springer, Heidelberg, Sept. 2014, pp. 358–379.

[58] T. K. FREDERIKSEN, M. KELLER, E. ORSINI, AND P. SCHOLL, *A unified approach to MPC with preprocessing using OT*, in ASIACRYPT 2015, Part I, T. Iwata and J. H. Cheon, eds., vol. 9452 of LNCS, Springer, Heidelberg, Nov. / Dec. 2015, pp. 711–735.

[59] J. FURUKAWA, Y. LINDELL, A. NOF, AND O. WEINSTEIN, *High-throughput secure three-party computation for malicious adversaries and an honest majority*, in EUROCRYPT 2017, Part II, J. Coron and J. B. Nielsen, eds., vol. 10211 of LNCS, Springer, Heidelberg, Apr. / May 2017, pp. 225–255.

[60] C. GENTRY, *A fully homomorphic encryption scheme*, PhD thesis, Stanford University, 2009.

[61] ——, *Fully homomorphic encryption using ideal lattices*, in 41st ACM STOC, M. Mitzenmacher, ed., ACM Press, May / June 2009, pp. 169–178.

[62] O. GOLDREICH, *The Foundations of Cryptography - Volume 2, Basic Applications*, Cambridge University Press, 2004.

[63] O. GOLDREICH AND L. A. LEVIN, *A hard-core predicate for all one-way functions*, in 21st ACM STOC, ACM Press, May 1989, pp. 25–32.

[64] O. GOLDREICH, S. MICALI, AND A. WIGDERSON, *How to play any mental game or A completeness theorem for protocols with honest majority*, in 19th ACM STOC, A. Aho, ed., ACM Press, May 1987, pp. 218–229.

[65] S. GOLDWASSER AND Y. LINDELL, *Secure multi-party computation without agreement*, Journal of Cryptology, 18 (2005), pp. 247–287.

[66] S. D. GORDON, S. RANELLUCCI, AND X. WANG, *Secure computation with low communication from cross-checking*, IACR Cryptology ePrint Archive, 2018 (2018), p. 216.

[67] S. HALEVI, C. HAZAY, A. POLYCHRONIADOU, AND M. VENKITASUBRAMANIAM, *Round-optimal secure multi-party computation*, in CRYPTO 2018, Part II, H. Shacham and A. Boldyreva, eds., vol. 10992 of Lecture Notes in Computer Science, Springer, 2018, pp. 395–424.

[68] C. HAZAY, E. ORSINI, P. SCHOLL, AND E. SORIA-VAZQUEZ, *Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT)*, in ASIACRYPT 2018, Part III, T. Peyrin and S. D. Galbraith, eds., vol. 11274 of LNCS, Springer, Dec. 2018, pp. 86–117.

[69] ——, *TinyKeys: A new approach to efficient multi-party computation*, in CRYPTO 2018, Part III, H. Shacham and A. Boldyreva, eds., vol. 10993 of LNCS, Springer, Aug. 2018, pp. 3–33.

[70] C. HAZAY, P. SCHOLL, AND E. SORIA-VAZQUEZ, *Low cost constant round MPC combining BMR and oblivious transfer*, in ASIACRYPT 2017, Part I, T. Takagi and T. Peyrin, eds., vol. 10624 of LNCS, Springer, Heidelberg, Dec. 2017, pp. 598–628.

[71] Y. HUANG, J. KATZ, AND D. EVANS, *Efficient secure two-party computation using symmetric cut-and-choose*, in CRYPTO 2013, Part II, R. Canetti and J. A. Garay, eds., vol. 8043 of LNCS, Springer, Heidelberg, Aug. 2013, pp. 18–35.

[72] R. IMPAGLIAZZO, L. A. LEVIN, AND M. LUBY, *Pseudo-random generation from one-way functions (extended abstracts)*, in 21st ACM STOC, ACM Press, May 1989, pp. 12–24.

[73] R. IMPAGLIAZZO AND S. RUDICH, *Limits on the provable consequences of one-way permutations*, in 21st ACM STOC, ACM Press, May 1989, pp. 44–61.

[74] Y. ISHAI, J. KILIAN, K. NISSIM, AND E. PETRANK, *Extending oblivious transfers efficiently*, in CRYPTO 2003, D. Boneh, ed., vol. 2729 of LNCS, Springer, Heidelberg, Aug. 2003, pp. 145–161.

[75] Y. ISHAI, M. PRABHAKARAN, AND A. SAHAI, *Founding cryptography on oblivious transfer - efficiently*, in CRYPTO 2008, D. Wagner, ed., vol. 5157 of LNCS, Springer, Heidelberg, Aug. 2008, pp. 572–591.

[76] ——, *Secure arithmetic computation with no honest majority*, in TCC 2009, O. Reingold, ed., vol. 5444 of LNCS, Springer, Heidelberg, Mar. 2009, pp. 294–314.

[77] S. JHA, L. KRUGER, AND V. SHMATIKOV, *Towards practical privacy for genomic computation*, in 2008 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, May 2008, pp. 216–230.

[78] J. KATZ, U. MAURER, B. TACKMANN, AND V. ZIKAS, *Universally composable synchronous computation*, in TCC 2013, A. Sahai, ed., vol. 7785 of LNCS, Springer, Heidelberg, Mar. 2013, pp. 477–498.

[79] M. KELLER, E. ORSINI, D. ROTARU, P. SCHOLL, E. SORIA-VAZQUEZ, AND S. VIVEK, *Faster secure multi-party computation of AES and DES using lookup tables*, in ACNS 17, D. Gollmann, A. Miyaji, and H. Kikuchi, eds., vol. 10355 of LNCS, Springer, Heidelberg, July 2017, pp. 229–249.

[80] M. KELLER, E. ORSINI, AND P. SCHOLL, *MASCOT: Faster malicious arithmetic secure computation with oblivious transfer*, in ACM CCS 16, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, eds., ACM Press, Oct. 2016, pp. 830–842.

[81] M. KELLER AND A. YANAI, *Efficient maliciously secure multiparty computation for RAM*, in EUROCRYPT 2018, Part III, J. B. Nielsen and V. Rijmen, eds., vol. 10822 of LNCS, Springer, Heidelberg, Apr. / May 2018, pp. 91–124.

[82] J. KILIAN, *Founding cryptography on oblivious transfer*, in 20th ACM STOC, ACM Press, May 1988, pp. 20–31.

[83] V. KOLESNIKOV AND R. KUMARESAN, *Improved OT extension for transferring short secrets*, in CRYPTO 2013, Part II, R. Canetti and J. A. Garay, eds., vol. 8043 of LNCS, Springer, Heidelberg, Aug. 2013, pp. 54–70.

[84] V. KOLESNIKOV, P. MOHASSEL, AND M. ROSULEK, *FleXOR: Flexible garbling for XOR gates that beats free-XOR*, in CRYPTO 2014, Part II, J. A. Garay and R. Gennaro, eds., vol. 8617 of LNCS, Springer, Heidelberg, Aug. 2014, pp. 440–457.

[85] V. KOLESNIKOV, J. B. NIELSEN, M. ROSULEK, N. TRIEU, AND R. TRIFILETTI, *DUPLO: Unifying cut-and-choose for garbled circuits*, in ACM CCS 17, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds., ACM Press, Oct. / Nov. 2017, pp. 3–20.

[86] V. KOLESNIKOV AND T. SCHNEIDER, *Improved garbled circuit: Free XOR gates and applications*, in ICALP 2008, Part II, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, eds., vol. 5126 of LNCS, Springer, Heidelberg, July 2008, pp. 486–498.

[87] E. LARRAIA, E. ORSINI, AND N. P. SMART, *Dishonest majority multi-party computation for binary circuits*, in CRYPTO 2014, Part II, J. A. Garay and R. Gennaro, eds., vol. 8617 of LNCS, Springer, Heidelberg, Aug. 2014, pp. 495–512.

[88] Y. LINDELL, *Fast cut-and-choose based protocols for malicious and covert adversaries*, in CRYPTO 2013, Part II, R. Canetti and J. A. Garay, eds., vol. 8043 of LNCS, Springer, Heidelberg, Aug. 2013, pp. 1–17.

[89] Y. LINDELL, E. OXMAN, AND B. PINKAS, *The IPS compiler: Optimizations, variants and concrete efficiency*, in CRYPTO 2011, P. Rogaway, ed., vol. 6841 of LNCS, Springer, Heidelberg, Aug. 2011, pp. 259–276.

[90] Y. LINDELL AND B. PINKAS, *An efficient protocol for secure two-party computation in the presence of malicious adversaries*, in EUROCRYPT 2007, M. Naor, ed., vol. 4515 of LNCS, Springer, Heidelberg, May 2007, pp. 52–78.

[91] ——, *A proof of security of Yao's protocol for two-party computation*, Journal of Cryptology, 22 (2009), pp. 161–188.

[92] ——, *Secure two-party computation via cut-and-choose oblivious transfer*, in TCC 2011, Y. Ishai, ed., vol. 6597 of LNCS, Springer, Heidelberg, Mar. 2011, pp. 329–346.

[93] Y. LINDELL, B. PINKAS, N. P. SMART, AND A. YANAI, *Efficient constant round multi-party computation combining BMR and SPDZ*, in CRYPTO 2015, Part II, R. Gennaro and M. J. B. Robshaw, eds., vol. 9216 of LNCS, Springer, Heidelberg, Aug. 2015, pp. 319–338.

[94] Y. LINDELL AND B. RIVA, *Cut-and-choose Yao-based secure computation in the online/offline and batch settings*, in CRYPTO 2014, Part II, J. A. Garay and R. Gennaro, eds., vol. 8617 of LNCS, Springer, Heidelberg, Aug. 2014, pp. 476–494.

[95] ——, *Blazing fast 2PC in the offline/online setting with security for malicious adversaries*, in ACM CCS 15, I. Ray, N. Li, and C. Kruegel:, eds., ACM Press, Oct. 2015, pp. 579–590.

[96] Y. LINDELL, N. P. SMART, AND E. SORIA-VAZQUEZ, *More efficient constant-round multi-party computation from BMR and SHE*, in TCC 2016-B, Part I, M. Hirt and A. D. Smith, eds., vol. 9985 of LNCS, Springer, Heidelberg, Oct. / Nov. 2016, pp. 554–581.

[97] D. MALKHI, N. NISAN, B. PINKAS, Y. SELLA, ET AL., *Fairplay-secure two-party computation system.*, in USENIX Security Symposium, vol. 4, San Diego, CA, USA, 2004, p. 9.

[98] A. MAY AND I. OZEROV, *On computing nearest neighbors with applications to decoding of binary linear codes*, in EUROCRYPT 2015, Part I, E. Oswald and M. Fischlin, eds., vol. 9056 of LNCS, Springer, Heidelberg, Apr. 2015, pp. 203–228.

[99] P. MOHASSEL AND M. FRANKLIN, *Efficiency tradeoffs for malicious two-party computation*, in PKC 2006, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, eds., vol. 3958 of LNCS, Springer, Heidelberg, Apr. 2006, pp. 458–473.

[100] P. MOHASSEL AND B. RIVA, *Garbled circuits checking garbled circuits: More efficient and secure two-party computation*, in CRYPTO 2013, Part II, R. Canetti and J. A. Garay, eds., vol. 8043 of LNCS, Springer, Heidelberg, Aug. 2013, pp. 36–53.

[101] P. MOHASSEL, M. ROSULEK, AND Y. ZHANG, *Fast and secure three-party computation: The garbled circuit approach*, in ACM CCS 15, I. Ray, N. Li, and C. Kruegel:, eds., ACM Press, Oct. 2015, pp. 591–602.

[102] J. MULLLIN, *Newegg trial: crypto legend takes the stand, goes for knockout patent punch*. Ars Technica, Nov. 24, 2013.

[103] M. NAOR, B. PINKAS, AND R. SUMNER, *Privacy preserving auctions and mechanism design*, in Proceedings of the 1st ACM conference on Electronic commerce, ACM, 1999, pp. 129–139.

[104] J. B. NIELSEN, P. S. NORDHOLT, C. ORLANDI, AND S. S. BURRA, *A new approach to practical active-secure two-party computation*, in CRYPTO 2012, R. Safavi-Naini and R. Canetti, eds., vol. 7417 of LNCS, Springer, Heidelberg, Aug. 2012, pp. 681–700.

[105] J. B. NIELSEN AND C. ORLANDI, *LEGO for two-party secure computation*, in TCC 2009, O. Reingold, ed., vol. 5444 of LNCS, Springer, Heidelberg, Mar. 2009, pp. 368–386.

[106] J. B. NIELSEN AND S. RANELLUCCI, *On the computational overhead of MPC with dishonest majority*, in PKC 2017, Part II, S. Fehr, ed., vol. 10175 of LNCS, Springer, Heidelberg, Mar. 2017, pp. 369–395.

[107] J. B. NIELSEN, T. SCHNEIDER, AND R. TRIFILETTI, *Constant round maliciously secure 2pc with function-independent preprocessing using lego*, in 24th NDSS Symposium, The Internet Society, 2017.
`http://eprint.iacr.org/2016/1069`.

[108] ——, *Constant round maliciously secure 2PC with function-independent preprocessing using LEGO*, in NDSS 2017, The Internet Society, Feb. / Mar. 2017.

[109] J. PERRY, D. GUPTA, J. FEIGENBAUM, AND R. N. WRIGHT, *Systematizing secure computation for research and decision support*, in SCN 14, M. Abdalla and R. D. Prisco, eds., vol. 8642 of LNCS, Springer, Heidelberg, Sept. 2014, pp. 380–397.

[110] K. PIETRZAK, *Subspace LWE*, in TCC 2012, R. Cramer, ed., vol. 7194 of LNCS, Springer, Heidelberg, Mar. 2012, pp. 548–563.

[111] B. PINKAS, T. SCHNEIDER, N. P. SMART, AND S. C. WILLIAMS, *Secure two-party computation is practical*, in ASIACRYPT 2009, M. Matsui, ed., vol. 5912 of LNCS, Springer, Heidelberg, Dec. 2009, pp. 250–267.

[112] E. PRANGE, *The use of information sets in decoding cyclic codes*, IRE Trans. Information Theory, 8 (1962), pp. 5–9.

[113] M. O. RABIN, *How to exchange secrets with oblivious transfer, 1981*, Harvard Center for Research in Computer Technology, Cambridge, MA, (1981).

[114] T. RABIN AND M. BEN-OR, *Verifiable secret sharing and multiparty protocols with honest majority (extended abstract)*, in 21st ACM STOC, ACM Press, May 1989, pp. 73–85.

[115] S. RAZI, *Nahj al-balagha*, Tenth century.

[116] O. REGEV, *On lattices, learning with errors, random linear codes, and cryptography*, J. ACM, 56 (2009), pp. 34:1–34:40.

[117] P. RINDAL, *libOTe: an efficient, portable, and easy to use Oblivious Transfer Library*. https://github.com/osu-crypto/libOTe.

[118] P. RINDAL AND M. ROSULEK, *Faster malicious 2-party secure computation with online/offline dual execution*, in USENIX, 2016, pp. 297–314.

[119] P. ROGAWAY, *The moral character of cryptographic work*. Cryptology ePrint Archive, Report 2015/1162, 2015. https://eprint.iacr.org/2015/1162.

[120] M.-J. O. SAARINEN, *Linearization attacks against syndrome based hashes*, in INDOCRYPT 2007, K. Srinathan, C. P. Rangan, and M. Yung, eds., vol. 4859 of LNCS, Springer, Heidelberg, Dec. 2007, pp. 1–9.

[121] T. SCHNEIDER AND M. ZOHNER, *GMW vs. Yao? Efficient secure two-party computation with low depth circuits*, in FC 2013, A.-R. Sadeghi, ed., vol. 7859 of LNCS, Springer, Heidelberg, Apr. 2013, pp. 275–292.

[122] A. SHAMIR, *How to share a secret*, Communications of the Association for Computing Machinery, 22 (1979), pp. 612–613.

[123] A. SHELAT AND C.-H. SHEN, *Two-output secure computation with malicious adversaries*, in EUROCRYPT 2011, K. G. Paterson, ed., vol. 6632 of LNCS, Springer, Heidelberg, May 2011, pp. 386–405.

[124] ——, *Fast two-party secure computation with minimal assumptions*, in ACM CCS 13, A.-R. Sadeghi, V. D. Gligor, and M. Yung, eds., ACM Press, Nov. 2013, pp. 523–534.

[125] J. STERN, *A method for finding codewords of small weight*, in Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings, 1988, pp. 106–113.

[126] S. R. TATE AND K. XU, *On garbled circuits and constant round secure function evaluation*, CoPS Lab, University of North Texas, Tech. Rep, 2 (2003), p. 2003.

[127] D. WAGNER, *A generalized birthday problem*, in CRYPTO 2002, M. Yung, ed., vol. 2442 of LNCS, Springer, Heidelberg, Aug. 2002, pp. 288–303.

[128] X. WANG, S. RANELLUCCI, AND J. KATZ, *Authenticated garbling and efficient maliciously secure two-party computation*, in ACM CCS 17, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds., ACM Press, Oct. / Nov. 2017, pp. 21–37.

[129] ——, *Global-scale secure multiparty computation*, in ACM CCS 17, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds., ACM Press, Oct. / Nov. 2017, pp. 39–56.

[130] S. WIESNER, *Conjugate coding*, ACM Sigact News, 15 (1983), pp. 78–88.

[131] A. C.-C. YAO, *Protocols for secure computations (extended abstract)*, in 23rd FOCS, IEEE Computer Society Press, Nov. 1982, pp. 160–164.

[132] ——, *How to generate and exchange secrets (extended abstract)*, in 27th FOCS, IEEE Computer Society Press, Oct. 1986, pp. 162–167.

[133] S. ZAHUR, M. ROSULEK, AND D. EVANS, *Two halves make a whole - reducing data transfer in garbled circuits using half gates*, in EUROCRYPT 2015, Part II, E. Oswald and M. Fischlin, eds., vol. 9057 of LNCS, Springer, Heidelberg, Apr. 2015, pp. 220–250.

[134] R. ZHU AND Y. HUANG, *JIMU: Faster LEGO-based secure computation using additive homomorphic hashes*, in ASIACRYPT 2017, Part II, T. Takagi and T. Peyrin, eds., vol. 10625 of LNCS, Springer, Heidelberg, Dec. 2017, pp. 529–572.